Predictive Resource Management for Scientific Workflows

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
Humboldt-Universität zu Berlin

von

M. Sc. Carl Philipp Witt

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. habil. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

GutachterInnen:
1. Prof. Dr.-Ing. Ulf Leser, Humboldt-Universität zu Berlin
2. Prof. Dr. habil. Odej Kao, Technische Universität Berlin
3. Prof. Dr. Ewa Deelman, University of Southern California

Tag der mündlichen Prüfung: 5. Juni 2020

Zusammenfassung

Wissenschaftliche Experimente produzieren Daten in nie dagewesenem Ausmaß. Um Erkenntnisse aus großen Mengen Rohdaten zu gewinnen, sind komplexe Datenanalysen erforderlich. Scientific Workflows sind ein Ansatz zur Umsetzung solcher Datenanalysen. Um Skalierbarkeit zu erreichen, setzen die meisten Workflow-Management-Systeme auf bereits existierende Lösungen zur Verwaltung verteilter Ressourcen, etwa Batch-Scheduling-Systeme. Workflow-Management-Systeme sind bereits in der Lage, die Allokation von Ressourcen automatisch und transparent mit verschiedenen Ressourcenmanagern abzuwickeln. Die Abschätzung der Ressourcen, die zur Ausführung einzelner Arbeitsschritte benötigt werden, wird aber immer noch an die Nutzer delegiert. Dies schränkt die Leistung und Benutzerfreundlichkeit von Workflow-Management-Systemen ein, da den Benutzern oft die Zeit, das Fachwissen oder die Anreize fehlen, den Ressourcenverbrauch genau abzuschätzen. Diese Arbeit untersucht, wie die Ressourcennutzung während der Ausführung von Workflows automatisch erlernt werden kann. Im Gegensatz zu früheren Arbeiten werden Scheduling und Vorhersage von Ressourcenverbrauch in einem engeren Zusammenhang betrachtet. Dies bringt verschiedene Herausforderungen mit sich, wie die Quantifizierung der Auswirkungen von Vorhersagefehlern auf die Systemleistung. Die wichtigsten Beiträge dieser Arbeit sind:

1. Eine Literaturübersicht aktueller Ansätze zur Vorhersage von Spitzenspeicherverbrauch mittels maschinellen Lernens im Kontext von Batch-Scheduling-Systemen.

2. Ein Scheduling-Verfahren, das statistische Methoden verwendet, um vorherzusagen, welche Scheduling-Entscheidungen verbessert werden können. Das Verfahren lieferte bis zu 40% kürzere Ausführungspläne als ein etabliertes Verfahren.

3. Ein Ansatz zur Nutzung von zur Laufzeit gemessenem Spitzenspeicherverbrauch in Vorhersagemodellen, die die fortwährende Optimierung der Ressourcenallokation erlauben. Umfangreiche Simulationsexperimente geben Einblicke in Schlüsseleigenschaften von Scheduling-Heuristiken und Vorhersagemodellen. Im Vergleich mit statischer Ressourcenallokation ergeben sich bis zu 76% reduzierte Laufzeiten.

4. Ein Vorhersagemodell, das die asymmetrischen Kosten überschätzten und unterschätzten Speicherverbrauchs berücksichtigt, sowie die Folgekosten von Vorhersagefehlern einbezieht. Der Ansatz wurde auf Hochenergiephysik-Workflows aus der Praxis evaluiert und reduzierte die Speicherverschwendung um 50%.

Diese Arbeit leistet Beiträge im Bereich der adaptiven und automatischen Allokation von Ressourcen in Workflow-Management-Systemen. Sie haben das Potenzial, den Benutzungsaufwand und die Leistung von Workflow-Management-Systemen wesentlich zu verbessern.

Abstract

Scientific experiments produce data at unprecedented volumes and resolutions. For the extraction of insights from large sets of raw data, complex analysis workflows are necessary. Scientific workflows enable such data analyses at scale. To achieve scalability, most workflow management systems are designed as an additional layer on top of distributed resource managers, such as batch schedulers or distributed data processing frameworks. Workflow management systems are already capable of automatically negotiating resources with different resource managers. They do not, however, automatically determine the amount of resources required for executing individual tasks in a workflow. The status quo is that workflow management systems delegate the challenge of estimating resource usage to the user. This limits the performance and ease-of-use of scientific workflow management systems, as users often lack the time, expertise, or incentives to estimate resource usage accurately.

This thesis is an investigation of how to learn and predict resource usage during workflow execution. In contrast to prior work, an integrated perspective on prediction and scheduling is taken, which introduces various challenges, such as quantifying the effects of prediction errors on system performance. The main contributions are:

1. A survey of peak memory usage prediction in batch processing environments. It provides an overview of prior machine learning approaches, commonly used features, evaluation metrics, and data sets.

2. A static workflow scheduling method that uses statistical methods to predict which scheduling decisions can be improved. The approach outperforms a state-of-the-art scheduling method on random graphs by producing up to 40% shorter execution plans.

3. A feedback-based approach to scheduling and predictive resource allocation, which is extensively evaluated using simulation. The results provide insights into the desirable characteristics of scheduling heuristics and prediction models. Compared to fixed resource allocation, feedback-based allocation reduces execution times on synthetic workflows by up to 76%.

4. A prediction model that reduces memory wastage. The design takes into account the asymmetric costs of overestimation and underestimation, as well as follow up costs of prediction errors. The approach is evaluated in a case study comprising large-scale high-energy physics workflows, reducing memory wastage by 50%.

The proposed designs are essential steps towards adaptive and automatic resource allocation for workflow management systems. Automatic resource allocation has the potential to improve the ease-of-use and performance of workflow management systems.

Acknowledgments

This dissertation would not have been possible without my family, friends, and life partner, Mathias. They have been incredibly patient and supportive during the last years, during which this project was always on my mind. Thanks, Lucas, for immensely valuable and never-ending meetings.

I thank my supervisor Ulf Leser, who put a lot of trust in me by selecting me for this scientific journey. I had a fantastic $3\frac{1}{2}$ years exploring the realms of intriguing problems, rare knowledge, and the many facets of a job in science. It was a pleasure and an honor to work with my talented co-workers, and all the other wonderful people who engage in scientific endeavors. Thank you, Ulf, for creating a research group with a supportive and inspiring atmosphere.

I would also like to thank the DFG graduate school SOAMED for providing amazing conditions for my research. I was given funding, feedback, and plenty of opportunities for exchange with other doctoral students. Especially the regular retreats have been helpful to structure years of work.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Contributions . . . 3
- 1.3. Thesis Content and Structure . . . 4
- 1.4. Published Material . . . 5

2. Fundamentals **7**
- 2.1. Scientific Workflow Management . . . 7
  - 2.1.1. Goals and Characteristics . . . 7
  - 2.1.2. Workflow Representation . . . 9
  - 2.1.3. Workflow Management Systems . . . 13
- 2.2. Scheduling . . . 16
  - 2.2.1. Static Task Graph Scheduling . . . 16
  - 2.2.2. Dynamic Workflow Scheduling . . . 20
- 2.3. Batch Execution of Scientific Workflows . . . 21
  - 2.3.1. Batch Schedulers . . . 21
  - 2.3.2. Batch Execution Model . . . 23
  - 2.3.3. Task Sizing . . . 24
- 2.4. Memory Usage Prediction . . . 27

3. Level-Order Sampling for Static Task Graph Scheduling **33**
- 3.1. Method . . . 33
  - 3.1.1. L-Order Sampling . . . 34
  - 3.1.2. Improvement Probability . . . 36
  - 3.1.3. Exploiting Improvement Probabilities Across Regions . . . 38
  - 3.1.4. Balancing Quality and Quantity of Improvements . . . 40
- 3.2. Experimental Results . . . 40
  - 3.2.1. Input Instances . . . 42
  - 3.2.2. Baseline Ranking Methods . . . 42
  - 3.2.3. Makespan Reductions . . . 43
  - 3.2.4. Progress and Convergence . . . 45
  - 3.2.5. Non-significant Factors . . . 48
- 3.3. Related Work . . . 48
- 3.4. Discussion . . . 49

4. Feedback-Based Scheduling of Scientific Workflows **53**
4.1. Method . . . 54
4.1.1. Scheduling . . . 54
4.1.2. Online Peak Memory Usage Prediction . . . 57
4.2. Experimental Design . . . 59
4.2.1. Synthetic Workflows . . . 59
4.2.2. Simulation Parameters . . . 63
4.3. Experimental Results . . . 64
4.3.1. Fixed Configurations . . . 64
4.3.2. Workflow-Specific Configuration . . . 69
4.4. Related Work . . . 74
4.5. Discussion . . . 76

5. Low-Wastage Regression for Peak Memory Prediction **79**
5.1. Method . . . 79
5.1.1. Memory Wastage Minimization . . . 80
5.1.2. Rectified Linear Allocation Model . . . 81
5.1.3. Constrained Optimization . . . 81
5.2. Case Study: IceCube Neutrino Observatory . . . 85
5.2.1. Scientific Workflows at IceCube . . . 85
5.2.2. Production Log Data . . . 86
5.3. Experimental Results . . . 88
5.3.1. Evaluation Approach . . . 88
5.3.2. Baseline Method . . . 89
5.3.3. Memory Allocation Quality . . . 90
5.3.4. Sensitivity to Initial Solutions . . . 93
5.3.5. Model Parameter Choices on the IceCube Data Set . . . 95
5.4. Related Work . . . 96
5.5. Discussion . . . 97

6. Conclusion **99**
6.1. Summary . . . 99
6.2. Challenges and Opportunities . . . 100
6.2.1. Prediction Accuracy . . . 100
6.2.2. Heuristics Design . . . 101
6.2.3. Qualitative Research . . . 102
6.2.4. Trust and Practicality . . . 103

Appendices **105**

A. Workflow Management Systems **107**
A.1. Workflow Management System Traits . . . 107
A.2. Common Workflow Management Systems . . . 110
A.3. Workflow Management Systems, Languages, and Tools . . . 112

B. Supplemental Figures **115**
B.1. Supplemental Figures for Chapter 3 115
B.2. Supplemental Figures for Chapter 4 116
B.3. Supplemental Figures for Chapter 5 118

C. Random Linear Models **119**

D. Workflow Partitioning **121**

Bibliography **125**

List of Figures **137**

List of Tables **141**

List of Algorithms **143**

List of Definitions **145**

1. Introduction

1.1. Motivation

Technology for obtaining, storing, and processing petabytes of data has changed science. Scientists now collect data at unprecedented resolutions and volumes, which can no longer be processed by a single computer. To extract insights from the data, scientists need scalable data analyses [Liew et al., 2017].

As an example, bioinformatics, material science, astronomy, and geoscience frequently require the processing of large data sets with various command-line tools arranged in a complex analysis pipeline. For instance, a recent study of the microbial diversity in the human gut analyzed approximately 37 trillion base pairs describing the genomes of thousands of microbial species [Almeida et al., 2019]. The analysis involved more than a dozen complex software packages and required several thousand years of CPU time. An example from radio astronomy is the recently acquired first image of a black hole. The Event Horizon Telescope produced petabytes of data that were reduced to megabyte-sized results using complex data processing workflows [Budrikis, 2019].

In the foreseeable future, connecting large numbers of computers is the only way to achieve petabyte storage and computing capacity. However, massive investments in hardware are not enough to enable data analyses at scale. Analyzing data on a distributed computing system requires several layers of complex logic. For instance, the design of algorithms that efficiently distribute computational work and move data between computers is challenging [Casanova et al., 2008]. Also, the data analyses themselves are complex and comprise many different tasks involving a variety of research software. Individual tasks produce intermediate results that require further processing by other tasks, leading to complex dependencies between different parts of the analysis.

The scientific workflow community conducts fundamental research on how to formalize, implement, debug, reproduce, and optimize data analyses at scale [Blankenberg et al., 2010, Albrecht et al., 2012, Wolstencroft et al., 2013, Ferreira da Silva et al., 2017, Di Tommaso et al., 2017]. This thesis develops methods to optimize the execution of scientific workflows through scheduling and prediction. Scheduling refers to the problem of deciding what parts of a workflow to execute on which machine and when [Drozdowski, 2010]. Prediction refers to the use of machine learning and statistical methods to make educated guesses about unknowns in the scheduling problem.

Scheduling can provide a variety of benefits, e. g., improved performance and increased user satisfaction, e. g., through fair distribution of waiting times among users. This thesis focuses on performance aspects, such as reducing workflow execution times. Good scheduling algorithms improve performance by balancing conflicting goals such as the distribution of

computation versus the locality of computation. Distribution of computation offers speedups through parallelization while locality of computation offers reductions in data transfer times. Ideally, a scheduler organizes work in a way that distributes work evenly while reducing the pressure on bottleneck resources.

Scheduling is a difficult problem, even without unknowns or uncertainty. A fundamental challenge in scheduling is the enormous number of possible solutions. Finding high-quality solutions is expensive because scheduling decisions that appear suboptimal can, in the long run, lead to optimality. State-of-the-art approaches to workflow scheduling fall into two categories [Chakravarthi and Vijayakumar, 2018]. Search approaches consider large numbers of schedules and tend to deliver high solution quality at high computational expense. Heuristics use intuitions about the properties of well-constructed schedules to restrict the number of considered scheduling decisions. This approach provides lower quality solutions at a lower computational expense.

In addition to its combinatorial complexity, obtaining planning information is another challenge for scheduling. An elemental aspect of scientific workflow scheduling is resource allocation, i. e., the amount of resources assigned to each task in a workflow [Juve, 2012]. Data centers for high-performance computing typically require reservations for their computational resources to manage the resource usage of different users. This forces users to estimate resource demands before the execution of a workflow [Tovar et al., 2018]. Users have to rely on their experience, laborious benchmarking, or trial-and-error to determine suitable amounts.

Replacing user estimates by statistical methods and machine learning is an opportunity to relief users from reasoning about technical details, and increase the accuracy of estimates by incorporating comprehensive historical data and data collected at run time. Providing more accurate planning information through predictions improves the quality and applicability of schedules [Agullo et al., 2016, Tsafrir et al., 2007, Gaussier et al., 2015] and learned heuristics may outperform manually designed heuristics [Mao et al., 2019]. However, the integration of prediction models into schedulers and workflow execution engines comes with various challenges:

- Training data. Historical data to train prediction models may not always be available or become stale as workflows are developed further or are applied to different input data.

- Loss Functions. A central challenge is determining the relationship between prediction errors and their impact on system performance. Overestimation of task resource usage leads to excess reservations, which decreases a system's throughput. Reserving insufficient resources for tasks typically leads to task termination and a subsequent attempt with increased resources. Quantifying and balancing the adverse effects is non-trivial, as they depend on a variety of factors, such as the current supply and demand of resources and the previously mentioned long-term effects of scheduling decisions.

- Maintenance. Although prediction models may unburden users from providing resource usage estimates, fixing a problem with a prediction model requires considerably more effort than correcting a user estimate.

1.2. Contributions

This thesis advances the state of the art in static workflow scheduling, dynamic scheduling involving memory usage predictions, and machine learning for memory usage predictions.

1. Chapter 2 provides an overview of state-of-the-art approaches to predict the memory consumption of computational jobs. The focus lies on batch-scheduling environments where underpredictions lead to task termination and thus have asymmetric costs compared to overpredictions. The survey summarizes common features, methods, metrics, and the data sets used by state-of-the-art approaches.

2. Chapter 3 proposes the Level-Order Sampling (LOS) method that derives a search algorithm from a particular class of scheduling heuristics using a novel approach to divide the schedule search space into regions. The method uses statistical methods to predict which scheduling decisions can be improved by estimating the probability of finding a better schedule in a given region. The predictions are then used to iteratively improve the current solution until a time budget is exhausted. LOS produced up to 40% shorter execution plans compared to a state-of-the-art heuristic.

3. Scientific workflow management systems often delegate the decision of how much memory to reserve for a task to the user. Chapter 4 evaluates the potential and robustness of learning resource usage of tasks during the execution of a workflow. The system creates an online learning model for different groups of tasks and updates it using resource usage measurements collected at run time to improve resource usage predictions continuously. The chapter proposes a feedback-based approach that allows incorporating training data collection or prediction errors into scheduling decisions. In an extensive simulation study, different aspects of system performance are compared under fixed and online learned resource usage estimates. In addition, the chapter investigates interaction effects between prediction models and scheduling strategies. Compared to fixed resource allocation, simulated workflow execution times could be reduced by up to 76%.

4. Chapter 5 proposes a novel prediction model that minimizes the impact of prediction errors on workflow execution in terms of resource wastage rather than minimizing the prediction errors. The Low-Wastage Regression (LWR) model takes into account (1) that under-predicting memory usage typically has a much more severe impact on performance than over-predicting memory usage and (2) follow-up costs of out-of-memory failures in terms of future required attempts. The proposed model is aware of the impact of prediction errors on workflow execution and is thus a step towards practical integration of resource usage predictions into workflow management systems. In a case study with production data from high-energy physics workflows, resource wastage could be reduced by 50%.

In summary, the thesis provides two perspectives on integrating scheduling and prediction. Chapter 3 proposes a scheduling algorithm that uses statistical methods to predict which

Assumption	Chapter 3	Chapter 4	Chapter 5
Execution environment	dedicated	batch scheduler	batch scheduler
Modeled resources	processors	processors, memory	memory
Prediction model	oracle	off-the-shelf, online updates	custom, online updates
User estimates	*n.a.*	hypothetical	real
Scheduling	static	dynamic	*n.a.*
Evaluation criteria	makespan	makespan, utilization	utilization
Workflows	random	synthetic	real

Table 1.1.: Comparison of the assumptions made in the core contribution chapters.

scheduling decisions can be improved. In Chapters 4 and 5, machine learning is used to predict the amount of memory needed for individual tasks in a workflow.

1.3. Thesis Content and Structure

The remainder of this thesis comprises five chapters. This section summarizes their contents and shows how they build upon each other. It also summarizes their overarching assumptions regarding the execution environment, workflow management system, and prediction quality, as outlined in Table 1.1.

Chapter 2 introduces fundamental terms and notation used throughout this thesis. First, an overview of scientific workflows and scientific workflow management systems is given. Second, basic notions of workflow scheduling are introduced, which form the basis for the scheduling aspects of Chapter 3 and Chapter 4. Third, different assumptions of workflow execution are discussed, which are relevant for the experimental design and evaluation in Chapter 4 and Chapter 5. Finally, a literature review of state-of-the-art methods for predicting peak memory usage provides a comprehensive overview of related work for the machine learning model proposed in Chapter 5.

Chapter 3 presents an algorithm for static task graph scheduling. The method is inspired by list scheduling approaches, which first compute a priority for each task and then assign tasks to processors in order of their priority [Kwok and Ahmad, 1999]. The proposed method derives variations of a schedule by randomly modifying task priorities. The search process is driven by estimates of the probability of improving over a current solution when modifying priorities in a specific part of the workflow.

Chapter 3 adopts typical assumptions made in the literature on static task graph scheduling. The execution environment for the workflow comprises a set of heterogeneous processors that are exclusively dedicated to the execution of a single workflow. The execution times of tasks

in the workflow are assumed to be known (oracle prediction model) for every processor. An execution plan is computed before the execution of the workflow (static scheduling), and random workflows are used for evaluation.

Chapter 4 evaluates the potential of learning resource usage estimates during the execution of a workflow. The chapter presents a feedback-based workflow execution model in which task memory usage is monitored and modeled at run time. The experiments show that learning memory usage at run time outperforms fixed memory estimates provided before the execution of the workflow by a large margin, resulting in higher resource utilization and lower workflow execution times.

Chapter 4 deviates in several assumptions from Chapter 3. The experiments assume workflow execution that requires resource reservations, which is typically the case in multi-user environments. In contrast to the previous chapter, resource usage is not assumed to be known but needs to be estimated. The experimental evaluation uses realistic synthetic workflows and two models for generating hypothetical user estimates for tasks in the workflows. Online learning is used to predict peak memory usages based on peak memory measurements of tasks that already completed during the execution of a workflow.

Chapter 5 presents a novel machine learning model for estimating the peak memory usage of tasks in a workflow based on their input file sizes. Based on the insights gathered in Chapter 4, a cost function is defined that relates prediction errors to memory wastage for all future attempts to execute a task. This provides a more comprehensive assessment of the cost of prediction errors in terms of wasted resources, which is formalized as an optimization criterion.

The method is evaluated using real-world workflows from production log data from a scientific workflow management system used in the IceCube research project. The underlying assumptions in this chapter are identical to the premises in Chapter 4. The production log data contains actual user estimates of peak memory usage for tasks. By applying the proposed prediction model, the utilization of allocated memory has been improved from approximately 50% for user estimates to 75%.

Chapter 6 summarizes the results of this thesis and points out directions and concrete opportunities for future research.

1.4. Published Material

The following provides an overview of the thesis contents that are based on prior publications and clarifies the contributions of the co-authors. Where required from the publisher, a copyright note indicates which figures and tables have been re-used. The author of this thesis has written all manuscripts mentioned in the following.

The static task graph scheduling method presented in Chapter 3 was published in [Witt et al., 2018]. The authors contributed as follows: Carl Witt developed and implemented the

scheduling method, conducted the experimental evaluation, and analyzed the results. Sam Wheating implemented the competitor method that was used as a baseline in the evaluation. Ulf Leser supervised the work and provided feedback and suggestions for the manuscript.

Chapter 4 is based on preliminary experiments published in [Witt et al., 2019c]. However, the scheduling framework, the selection of scheduling methods, the prediction models, the experimental design, and the analysis of the results have been further developed and improved for this thesis. The authors contributed as follows: Carl Witt designed and implemented the feedback-based workflow execution mechanism, conducted the experimental evaluation, and analyzed the results. Dennis Wagner contributed early ideas for conservative regression, which have been replaced later. Ulf Leser supervised the work and provided feedback and suggestions for the manuscript.

The peak memory prediction method presented in Chapter 5 and its evaluation on the IceCube case study have been published in [Witt et al., 2019b]. The authors contributed as follows: Carl Witt designed and implemented the prediction model, analyzed the production log data, and conducted the experimental evaluation. Jakob van Santen extracted and prepared the production logs and contributed paragraphs in the background section of the manuscript. Ulf Leser supervised the work and provided feedback and suggestions for the manuscript.

A survey on predictive performance modeling was published in [Witt et al., 2019a]. The literature review on machine learning for memory usage prediction in Section 2.4 benefited from the overview gained during this literature review but contains new original material focusing on peak memory usage prediction, as compared to the general predictive performance modeling scope covered in our previous publication. The authors of the original publication contributed as follows: Carl Witt designed the comparison criteria and selected and reviewed the majority of papers. Marc Bux reviewed time-series related predictive performance modeling. Wladislaw Gusew reviewed black-box monitoring techniques for performance measurement. Ulf Leser supervised the work and provided feedback and suggestions for the manuscript and its revisions.

2. Fundamentals

This chapter introduces methods and notation upon which this thesis builds. It covers four areas: (1) scientific workflow languages and scientific workflow management systems, (2) workflow scheduling, (3) workflow execution in environments that require resource reservation, and (4) a survey of methods for memory usage prediction.

2.1. Scientific Workflow Management

A scientific workflow is a computer program that describes a data analysis or computational experiment in terms of tasks that exchange data. Scientific workflow languages serve the purpose of describing such data processing pipelines; scientific workflow management systems are used to execute workflow descriptions. Section 2.1.2 covers the fundamentals of workflow descriptions, and Section 2.1.3 provides and an overview of the current landscape of scientific workflow management systems.

Definition 2.1 (Workflow)**.** *A workflow is defined by a set of tasks, dependencies between tasks, and a specification of the data resources that constitute the input and output of the workflow [Liew et al., 2017]. Tasks represent computation, and task dependencies define the flow of data between tasks.*

2.1.1. Goals and Characteristics

The benefits of implementing a data analysis or computational experiment using a scientific workflow language can be grouped into five categories.

- Diversity allows the experiment or analysis to use various software, programming languages, and data models. This is particularly important for complex analyses involving software developed by multiple parties.

- Automation eliminates human interactions from the experiment or analysis. This frees up human resources and is a prerequisite for achieving scalability and reproducibility.

- Portability allows an experiment or analysis to be executed in different computational environments. This entails different operating systems with different software environments as well as different compute sites managed by different resource managers or batch schedulers.

- Scalability allows the parallel execution of the tasks of an experiment or analysis, possibly on different machines or across geographically distributed compute sites.

- Reproducibility allows the experiment or analysis to be reconstructed precisely and executed repeatedly to reproduce its results.

Scientific workflows rely on four fundamental characteristics to achieve these benefits: task-based computing, translation, the black-box assumption, and idempotence.

Task-based computing refers to an approach to parallel and distributed programming where computation is divided into tasks that can be executed in parallel. In a scientific workflow, a task corresponds to an invocation of a program that requires no user interaction and is invoked via a command-line interface. A task may read data from files that have been produced by one or more tasks. Other than that, a task is self-sufficient in the sense that it does not communicate with other tasks. Finally, a task is not preemptable, i. e., its execution cannot be suspended and resumed elsewhere.

Definition 2.2 (Task)**.** *A task is a non-interactive, self-sufficient, non-preemptable invocation of a program using its command-line interface. The task reads from specified input files and writes to specified output files; its input files may have been produced by other tasks.*

Translation refers to the compilation or interpretation of a workflow description for the execution in a specific execution environment. This entails technical details such as the type and location of a file system or the mode of resource negotiation. A translation process is essential to achieve portability and also allows for automatic optimization, such as scheduling, estimating resource requirements, or even choosing between different implementations of a task. However, abstraction also limits expressiveness, e. g., some workflow languages do not allow for dynamic branching or iteration. Limited expressiveness also hinders manual performance optimization, as a user may not be able to express the most efficient way to solve a problem using a restricted set of language elements. On the other hand, the possibility to delegate technical details to the translation process has the potential to increase the ease of use significantly and to reduce the time needed to implement an analysis or experiment.

The black-box assumption refers to the absence of restrictions on the programs and files used in the analysis or experiment. Specifically, no assumptions are made on the language used to implement a program. The only requirement is a command-line interface that allows for automated interaction with the program. The only restrictions on file content and file format arise from the composed programs, which allows for the use of arbitrary data models.

Idempotence refers to the requirement that tasks are free of side-effects, i. e., starting the same task on the same inputs always produces the same outputs. This implies that programs are self-sufficient in completing their task, given their input files. This allows for distribution across different compute sites and helps to mitigate transient failures by means of restarting tasks.

Comparison to Distributed Computing Application Programming Interfaces

A vast landscape of paradigms and frameworks for distributing computing exists, which are potentially competing approaches to implement large scale data analyses and experiments. The closest approaches to scientific workflows are distributed computing application programming interfaces like Spark [Zaharia et al., 2012] or Dask [Rocklin, 2015]. Similar to the scientific workflow approach, these approaches use tasks to parallelize work and allow for graph structured dependencies between tasks. The unique selling point of scientific workflows is their focus on integrating diverse software into an analysis or experiment.

While it is certainly possible to call arbitrary external programs in many distributed computing frameworks, distributing computing application programming interfaces like Spark or Dask rely on specific data models like arrays or sets with distributed implementations such as the resilient distributed data set in Spark. These data models restrict the format in which data is exchanged between tasks. Scientific workflows on the other hand make a black box assumption which allows for the use of complex and domain specific data and file formats, e. g., for satellite images or genomics data. This requirement also makes it difficult to express complex data analyses in terms of distributed computing approaches like MapReduce [Dean and Ghemawat, 2008], which is centered around the key-value data model, or the message passing interface [Gropp et al., 1999], which is centered around specific communication primitives between parallel processes.

In summary, the focus on the composition of programs sets scientific workflows apart from other distributed computing paradigms. This naturally allows for the integration of diverse software and programming languages. An advantage of distributed computing application programming interfaces over workflows is typically performance, as computation and communication can be specified on a lower level of abstraction. This, however, also significantly increases the development costs of a workflow.

2.1.2. Workflow Representation

This section introduces the concept of workflow languages and language independent workflow representation in the form of a directed acyclic graph.

Workflow Languages

A workflow language provides abstractions for solutions of common problems in scientific workflows. A workflow management system (see Section 2.1.3) provides the software to execute a workflow defined using these abstractions. Learning a workflow language constitutes an investment with the benefit of gaining access to ready-made, tested, and optimized implementations of these abstractions. Three central abstractions for scientific workflows are dependencies between tasks, failure handling, and task execution environments, as explained in the following.

Definition 2.3 (Workflow Language)**.** *A workflow language is the means by which executable workflow descriptions are expressed. A workflow language may be implemented as a domain-specific language or an application programming interface.*

Arguably the central abstraction in every workflow language is the composition of programs through dependent tasks. A dependency between two tasks indicates that one task produces the files that the other task processes. A dependency between tasks implies that they have to be executed in sequence, and the (transitive) absence of dependencies indicates that tasks can be executed in parallel. Various approaches to defining tasks and their dependencies exist:

- Explicit construction of a directed acyclic graph. Tasks correspond to vertices, and dependencies are expressed through directed arcs between vertices. The graph can be constructed using, for instance, an application programming interface for different host programming languages.

- Rule-based approaches. A task corresponds to the application of a rule that defines how to produce output files from input files. This is similar to how software like *make* derives an acyclic directed graph from a set of rules. File names may contain wildcard characters, allowing the rule set to generate various directed acyclic graphs. Based on which output files are desired, only part of the graph can be constructed and executed.

- Processes and channels. Tasks are instances of processes that are connected to input and output channels from which they receive input files and emit output files, respectively. Channels serve as communication queues that pass files between processes and can be composed via operators into another channel, e. g., to implement operations like forking, joining, or pairing files from different channels. This allows for tasks that are created dynamically at run time, based on the output of other tasks, and thus allows for powerful constructs like feedback loops.

- Functional programming. The Cuneiform workflow language [Brandt et al., 2015] is based on functional programming, where tasks correspond to function evaluations, and dependencies are expressed through function composition. In this setting, a dependency takes the form of a nested expression where the inner expression has to be evaluated before the outer expression.

A second abstraction concerns failure handling. Tasks can fail for various reasons, either transiently or permanently. A standard mechanism to deal with transient failures is to restart the task, assuming that the cause of the problem may have vanished. If a task has failed due to a specific reason such as insufficient compute resources, a dedicated failure handling strategy might be used, such as doubling the allocated resources. Several workflow languages provide constructs to express failure handling strategies for a task or a group of tasks and leave it to the workflow management system to maintain state for each task and take according action, such as changing allocated resources or retrying execution. This also implies that a

task in a workflow can correspond to a series of task attempts where dependencies refer only to the final, successful attempt of that sequence.

A third central abstraction in scientific workflows concerns the physical compute environment in which a task is executed. Tasks in the workflow are defined against an abstract environment that makes the underlying physical resources exchangeable. For instance, the Pegasus workflow management system uses the concept of logical file identifiers at the level of the workflow description. At the execution level, the logical file identifiers are mapped to physical paths. Workflow languages typically also have language elements that allow specifying constraints on the task execution environment, such as the amount of required CPU cores, memory, and execution time.

Currently, a wide range of workflow languages and management systems is available (Section 2.1.3). In the language domain, different standardization efforts have recently been undertaken, such as the Common Workflow Language [Amstutz et al., 2016]. However, establishing a standard seems challenging due to the wide range of use cases and requirements that exist in practice.

Directed Acyclic Graphs

This thesis takes an intermediate perspective between workflow language and the physical level. Workflows are represented as directed acyclic graphs, which is a language-independent representation of tasks and their dependencies. Technical aspects, such as mapping between logical file identifiers and physical file paths, are not considered, as they are irrelevant to predictive resource allocation. This section recapitulates the necessary graph theoretic and workflow-specific definitions.

Different options to describe a workflow through a directed acyclic graph exist. This thesis uses a rather high-level representation, where tasks are represented as the vertices of a graph. A directed edge between two vertices indicates that one task has to be executed entirely before the other.

Definition 2.4 (Workflow Graph)**.** *A workflow is represented using a directed acyclic graph* $G = (V, E)$ *that models tasks* $V = \{T_1, T_2, ..., T_n\}$ *and their dependency relationships* $E \subset V \times V$*.*

This work assumes that every task in a workflow has an abstract task label. Abstract task labels group the tasks in a workflow into sets of tasks with potentially similar resource usage. This will be of relevance in Chapter 4 and Chapter 5, where a prediction model for each abstract task is trained during the execution of the workflow.

Definition 2.5 (Abstract Task)**.** *Let* $G = (V, E)$ *be a workflow graph. An abstract task* A *is a set of tasks* $A \subseteq V$ *that indicates similarities in the resource usage of tasks. It is assumed that every task belongs to exactly one abstract task.*

Given a workflow graph $G = (V, E)$, the transitive closure of the precedence constraints E gives a strict partial order of the elements of V denoted as $\prec$. $T_i \prec T_j$ denotes that there is a path from T_i to T_j. It is assumed that at most one directed edge exists between two tasks, i. e., multigraphs are not allowed.

Definition 2.6 (Path)**.** *A path in a directed acyclic graph $G = (V, E)$ is a sequence of tasks $T_i, \ldots, T_k \in V$ such that $(T_j, T_{j+1}) \in E, i \leq j < k$.*

If there is a path from T_i to T_j, T_j is called a descendant of T_i. Equivalently, $T_j \succ T_i$ denotes that T_i is an ancestor of T_j. If neither $T_i \prec T_j$ nor $T_i \succ T_j$, the tasks are called incomparable, which is denoted with $T_i || T_j$. In the context of scheduling, $T_i \prec T_j$ means that T_i has to finish execution before T_j can start. $T_i || T_j$ means that T_i and T_j can be executed in parallel. A task without descendants is called an exit task.

Definition 2.7 (Entry/Exit Task)**.** *Let $G = (V, E)$ be a directed acyclic graph. The indegree of a task T_i equals the number of edges that point to that task $|\{(T_a, T_b) \in E \mid T_b = T_i\}|$. An entry task is a task with zero indegree. Analogously, the outdegree equals $|\{(T_a, T_b) \in E \mid T_a = T_i\}|$. An exit task is a task with zero outdegree.*

Finally, the level of a task is a useful concept for several algorithms operating on workflows. In Chapter 3, a static task graph scheduling method is proposed that generates random topological orders by sampling permutations of tasks with equal level. In Chapter 4, a dynamic scheduling heuristic is evaluated that prioritizes tasks in proportion to their level. The tasks in a workflow can also be partitioned using their levels, which is helpful in deriving lower bounds on the execution time of a workflow (Appendix D).

Definition 2.8 (Level)**.** *The level of a task is defined as the length of a longest path from the task to an exit task. The level is defined recursively based on the levels of its descendants.*

$$\mathit{level}(T_i) = \begin{cases} 0 & T_i \text{ is an exit task} \\ 1 + \max\{\mathit{level}(T_j) \mid T_i \prec T_j\} & \text{otherwise} \end{cases} \tag{2.1}$$

Workflow State

During the execution of a workflow, the workflow information represented by a directed acyclic graph is complemented with state information. This comprises, for instance, the information which tasks have finished so far. This information can then be used to infer which tasks are ready for execution. The following task life cycle summarizes the basic execution semantics used in the simulation experiments in this thesis.

Definition 2.9 (Task Lifecycle)**.** *A task is initially in the state* submitted*. As soon as all of its predecessor tasks have finished successfully, it transitions to the* ready *state. If the task is in the* ready *state, its execution can be started. After the execution starts, the task transits to the* running *state. If its execution fails, the task transits back to the* ready *state. If execution succeeds, the task transits to the* finished *state.*

This implies that the execution of tasks that depend on each other can not overlap, i. e., a predecessor task has to finish successfully before the successor task can be started.

2.1.3. Workflow Management Systems

Dozens of scientific workflow management systems are available (see Appendix A). This section summarizes (1) common aspects used to characterize and compare scientific workflow management systems and (2) major concrete systems at the time of writing. This introduction is based on an analysis of 11 survey papers that compare workflow management systems according to different criteria. In addition, a list of workflow management systems [Amstutz et al., 2019] known to implement the Common Workflow Language [Amstutz et al., 2016] is used as a source. In total, the 12 sources cover 47 scientific workflow management systems, as shown in Table 2.1.

A scientific workflow management system is a piece of software that executes scientific workflows. Possible aspects to characterize scientific workflow management systems range from technical aspects like data access modes to usability aspects like the mode of user interaction. For a complete list of comparison criteria used by the 11 surveys, refer to Appendix A. Two commonly used comparison criteria across the surveys are platform support and container support.

Platform Support

A central comparison criterion is how and where scientific workflows can be executed. A workflow management system can be run as a standalone program with direct access to compute resources, or on top of a distributed computing platform. Distributed computing platforms include batch schedulers (see Section 2.3) like Slurm [Yoo et al., 2003] or HTCondor [Thain et al., 2005]. Examples of distributed computing frameworks include Hadoop [Vavilapalli et al., 2013] and Mesos [Hindman et al., 2011]. Which platforms are supported by a workflow management system is of central importance to users because their access to large-scale compute resources is often tied to the use of a specific platform, e. g., the resource manager used to operate an organization's compute cluster. Leveraging distributed computing platforms is also a prerequisite for executing large-scale workflows. Supporting a range of platforms contributes to the portability of workflows.

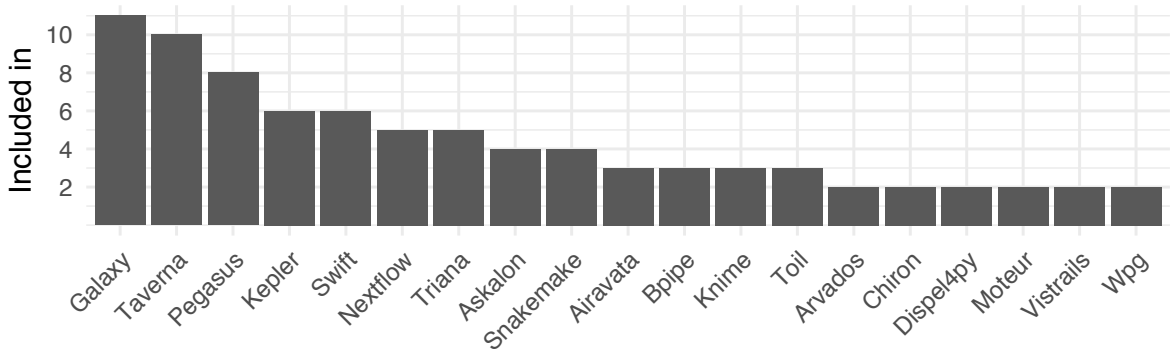

Figure 2.1.: Number of surveys from Table 2.1 that include a workflow management system for comparison against others. Only systems compared in more than one survey are shown.

Container support

For software deployment, state-of-the-art workflow management systems support the use of container engines. Container engines like Docker, Singularity, and Rkt simplify the software deployment through definition and bundling of software environments in a single file. This provides a light-weight alternative to virtual machines. Support of containerized tasks contributes to the goal of diversity, i. e., scientific workflows leveraging a wide range of software libraries and programming languages. It also helps reproducibility, as the software deployment process is typically intricate for complex workflows, and the software and its sources develop over time. Finally, container support also facilitates portability because it significantly reduces the time to make a workflow runnable in a different execution environment.

Concrete Systems

Figure 2.1 shows the number of sources from Table 2.1 in which each scientific workflow management system is compared to other workflow management systems. The figure includes only workflow management systems cited more than once. Appendix A.2 provides the complete list. The most popular systems with respect to being compared to other workflow management systems are Galaxy [Blankenberg et al., 2010], Taverna [Wolstencroft et al., 2013], and Pegasus [Deelman et al., 2015]. However, this citation-based metric is biased towards workflow management systems used in an academic context. In the industrial domain, workflow management systems like Luigi or Apache Airflow appear to be more popular, neither of which have an accompanying scientific publication. Although there seems to be a divide between academic and industrial users, this does not imply that the workflow management systems are fundamentally different. For instance, the Common Workflow Language is a standard that is adopted both by academic and industrial workflow management systems.

Reference	N	Scientific Workflow Management Systems Compared in Source
[Atkinson et al., 2017]	13	Airavata, Askalon, Dispel4py, Galaxy, Guse, Kepler, Knime, Moteur, Pegasus, Swift, Taverna, Triana, Wings
[Boulakia et al., 2017]	5	Galaxy, Nextflow, Openalea, Taverna, Vistrails
[Bux, 2017]	8	Cuneiform, Galaxy, Hi-WAY, Knime, Pegasus, Snakemake, Swift, Taverna
[Di Tommaso et al., 2017]	5	Bpipe, Galaxy, Nextflow, Snakemake, Toil
[Ferreira da Silva et al., 2017]	15	Adios, Airavata, Askalon, Bobolang, Dispel4py, Fireworks, Galaxy, Kepler, Makeflow, Moteur, Nextflow, Pegasus, Swift, Taverna, Triana
[Khan et al., 2017]	9	Askalon, Chiron, Galaxy, Kepler, Pegasus, Swift, Taverna, Triana, Wpg
[Leipzig, 2017]	15	Agave, Arvados, BigDataScript, Bpipe, Dnanexus, Galaxy, Luigi, Nextflow, Pegasus, Queue, Ruffus, Sevenbridges, Snakemake, Taverna, Toil
[Liew et al., 2017]	7	Airavata, Kepler, Knime, Meandre, Pegasus, Swift, Taverna
[Liu et al., 2015]	9	Askalon, Chiron, Galaxy, Kepler, Pegasus, Swift, Taverna, Triana, Wpg
[Mork et al., 2015]	7	Galaxy, Kepler, Pegasus, Rapidminer, Taverna, Triana, Vistrails
[Wang and Peng, 2019]	5	Bpipe, Galaxy, Nextflow, Snakemake, SoS
CWL Implementers List [Amstutz et al., 2019]	13	Airflow, Arvados, Awe, Calrissian, Consonance, Cwl-Tes, Cwlexec, Galaxy, Reana, Taverna, Toil, Xenon, Yacle

Table 2.1.: Surveys comparing scientific workflow management systems and the systems they compare. N denotes the number of compared systems in a survey.

2.2. Scheduling

2.2.1. Static Task Graph Scheduling

Static task graph scheduling is the problem of computing an execution plan for a workflow on a given infrastructure such that the total execution time is minimized. In the following, the problem is formally defined by introducing the terms infrastructure, schedule, and allocation function.

The problem is defined for a specific compute infrastructure that comprises processing elements, processors for short, that execute tasks and a network that allows for data transfers between processors. Processors handle one task at a time, i. e., the model does not take into account concepts like fractional resource utilization or resource contention due to parallel execution of tasks on a single processor. The standard network model is a fully connected network with heterogeneous data transfer rates between processors and no network contention [Kwok and Ahmad, 1999]. Tasks being executed on the same processor communicate for free, i. e., they can exchange any amount of data instantaneously.

Definition 2.10 (Infrastructure)**.** *An infrastructure $I = (P, b)$ specifies processors $P = \{p_1, p_2, ..., p_m\}$ and data transfer rates $b\colon P \times P \to \mathbb{R}^+$ between each pair of processors.*

In addition to an infrastructure, an input instance to the task graph scheduling problem comprises information about the workflow to be scheduled, namely the execution times $w(T_i, p_j)$ of each task on each processor and the amount of data $c(T_i, T_j)$ that needs to be transferred between tasks. There are no constraints other than non-negativity on the task run times. The general case where tasks have different run times on different processors is referred to as scheduling with heterogeneous resources. This information is summarized in a communication dag.

Definition 2.11 (Communication Dag)**.** *A communication dag is a directed acyclic graph $G = (V, E, w, c)$ that describes a workflow's tasks $V = \{T_1, T_2, ..., T_n\}$, the dependency relationships $E \subset V \times V$ between tasks, their execution duration on the different processors $w\colon V \times P \to \mathbb{R}^+$, and the amount of data to be transferred from a task to another $c\colon E \to \mathbb{R}^+$.*

A solution for an input instance is an execution plan that comprises an allocation function and a schedule. The allocation function specifies which task is executed on which processor. The notation is similar to the notation used in [Casanova et al., 2008].

Definition 2.12 (Allocation Function)**.** *Let $I = (P, b)$ be an infrastructure and $W = (V, E, w, c)$ a communication dag. An allocation function alloc$\colon V \to P$ maps tasks to processors.*

A schedule specifies the start times of tasks on their assigned processors. It ensures that dependencies between tasks are met, i. e., all predecessor tasks of a task have finished, and all data can be transferred in time.

Definition 2.13 (Schedule)**.** *Let I be an infrastructure, $W = (V, E, w, c)$ a communication dag, and alloc an allocation function. Let $p_i = \mathit{alloc}(T_i)$ and $p_j = \mathit{alloc}(T_j)$ be the processors assigned to tasks T_i, T_j, respectively. A schedule is a function $\sigma \colon V \to \mathbb{R}$ such that*

$$\forall T_i, T_j \colon T_i \prec T_j \Rightarrow \begin{cases} \sigma(T_i) + w(T_i, p_i) \leq \sigma(T_j) & p_i = p_j \\ \sigma(T_i) + w(T_i, p_i) + c(T_i, T_j)/b(p_i, p_j) \leq \sigma(T_j) & \textit{otherwise} \end{cases}$$

The quality of an execution plan is measured by its total execution time. This refers to the latest planned finish time across all tasks.

Definition 2.14 (Makespan)**.** *Let $W = (V, E, w, c)$ be a communication dag and I be an infrastructure. Let σ be a schedule and alloc an allocation function for W on I. The makespan $\mathit{MS}(\sigma, \mathit{alloc})$ of the execution plan is defined as the latest finish time of a task:*

$$\mathit{MS}(\sigma, \mathit{alloc}) = \max\{\sigma(T_i) + w(T_i, \mathit{alloc}(T_i)) \mid T_i \in V\}$$

Many variations of static task graph scheduling problem exist, e. g., making assumption about the graph topology, allowing tasks to be executed partially on one processor and finished on another, etc. [Kwok and Ahmad, 1999]. Minimizing the makespan of an execution plan is NP-complete for most variations of the problem, including the one presented here [Casanova et al., 2008]. This has fostered the development of a wide variety of scheduling heuristics.

Integrating these heuristics in production systems comes with a couple of challenges, such as obtaining the various information required for planning. Due to simplifying assumptions during scheduling, the execution plan may have to be modified for execution in a real system. On the other hand, it has been shown that static task graph scheduling can outperform myopic scheduling strategies, even when using inaccurate information in the planning stage [Agullo et al., 2016]. A method for static task graph scheduling is presented in Chapter 3.

List Scheduling and Task Weighting

List scheduling is a well-known approach for designing static task graph scheduling heuristics [Kwok and Ahmad, 1999]. The problem is divided into a task prioritization phase and a processor assignment phase. During the prioritization phase, an ordering of the tasks is computed that reflects the relative priorities of the tasks.

Definition 2.15 (Ordering)**.** *Let $W = (V, E, w, c)$ be a communication dag. An ordering of the tasks $V = \{T_1, \cdots, T_n\}$ is a permutation $\pi \colon V \to \{1, ..., n\}$. The value $\pi(T_i)$ is referred to as the rank of T_i.*

Some processor assignment methods require a topological ordering of the tasks. This ensures that when assigning a task a start time and a processor, all of its predecessors have already been scheduled.

Definition 2.16 (Topological Ordering)**.** *Let $W = (V, E, w, c)$ be a communication DAG and π an ordering of the tasks. π is a topological ordering if every task that has a higher rank than another task is either an ancestor of that task or can be executed in parallel with it:*

$$\forall T_i, T_j \in V \colon \pi(T_i) > \pi(T_j) \Rightarrow T_i \prec T_j \vee T_i || T_j$$

An ordering can be computed using a weighting scheme that assigns real numbers to the tasks reflecting their relative priority for scheduling. The weight could for instance represent an estimate of how critical the task is for advancing the execution of the dag. An ordering of the tasks can then be obtained by sorting the tasks according to their weight and, if necessary, breaking ties.

Definition 2.17 (Weighting Scheme)**.** *Let $W = (V, E, w, c)$ be a communication dag. A weighting scheme $V \to \mathbb{R}$ is a function that maps tasks to real numbers representing their priority.*

In the processor assignment phase, the scheduler considers the tasks in decreasing order of their rank to decide on which processor they should run. This can be done using greedy criteria such as choosing the processor that minimizes the finish time of the task. The next section gives an example of a weighting scheme and a processor assignment method.

Heterogeneous Earliest Finish Time Algorithm

The Heterogeneous Earliest Finish Time (HEFT) algorithm [Topcuoglu et al., 2002] is a highly-cited list scheduling method to solve the task graph scheduling problem. HEFT first averages the computation and communication costs over all available resources and then propagates combined computation and communication costs from the bottom of the dag to the entry tasks. This results in task weights that provide an estimate of the length of the critical path. The idea is to prioritize tasks on the critical path by assigning them to processors where they finish earliest, thereby exploiting the heterogeneity of the resources.

During the task prioritization phase, HEFT uses a weighting scheme that computes the weight of each task recursively based on the weights of its children. Let $I = (P, b)$ be an

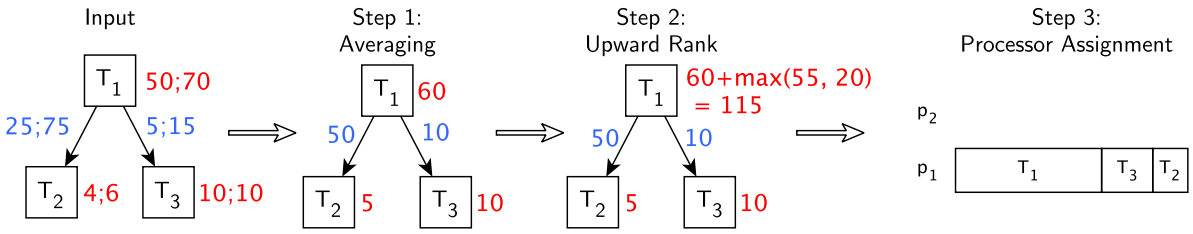

Figure 2.2.: Example of the HEFT algorithm on two processors p_1 and p_2. The computation times are annotated in red to the nodes as $w(T_i, p_1); w(T_i, p_2)$. The communication times are annotated in blue to the edges as $c(T_i, T_j)/b(p_1, p_2)$; $c(T_i, T_j)/b(p_2, p_1)$. In step 1, average costs are computed. In step 2, task priorities are determined. Step 3 determines the schedule by assigning each task to the processor that minimizes finish time.

infrastructure with m processors $P = \{p_1, \ldots, p_m\}$. Let $\bar{w}$ denote the computation costs averaged over all processors and $\bar{c}$ the communication costs averaged over all network links.

$$\bar{w}(T_i) = \frac{1}{m} \sum_{k=1}^{m} w(T_i, p_k) \tag{2.2}$$

$$\bar{c}(T_i, T_j) = \frac{c(T_i, T_j)}{m^2 - m} \sum_{1 \leq k,l \leq m, k \neq l} b(p_k, p_l) \tag{2.3}$$

Average computation and communication times approximate computation and communication costs independently of task placement. In the next step, the task dependencies are taken into account. Intuitively, tasks with more work ahead of them are prioritized. This is formalized by computing for each task the maximum length of a path that starts at this task, where the length of a path corresponds to the summed computation and communication costs. Since the path lengths can be conveniently computed in a bottom-up fashion, they are referred to as upward ranks of the tasks.

Definition 2.18 (Upward Rank)**.** *Let $W = (V, E, w, c)$ be a communication dag and $\bar{w}, \bar{c}$ average computation and communication costs, respectively. The upward rank of a task is defined recursively as*

$$\mathit{urank}(T_i) = \begin{cases} \bar{w}(T_i) & T_i \text{ is an exit task} \\ \bar{w}(T_i) + \max\{\bar{c}(T_i, T_j) + \mathit{urank}(T_j) \mid (T_i, T_j) \in E\} & \text{otherwise} \end{cases} \tag{2.4}$$

Processor Assignment

In the processor assignment phase, HEFT selects for each task, in order of decreasing upward rank (ties arbitrarily broken), the processor that minimizes the earliest finish time of the task. The earliest finish time of a task T_i on a given processor p_j depends on T_i's parents' finish

times, the data transfer times from the parent's scheduled processors to p_i, and T_i's execution time $w(T_i, p_j)$. When scheduling a task, all its parent tasks have already been scheduled, since sorting by decreasing upward rank yields a topological order. HEFT uses an insertion-based strategy that schedules a task T_i between tasks already scheduled on p_j if T_i's input data can be transferred to p_i in time, and the gap has a length of at least $w(T_i, p_j)$, i. e., the already scheduled tasks are not delayed.

Figure 2.2 shows an example of the HEFT method. Note how the processor assignment based on upward ranks allows HEFT to balance parallelization speedups with communication overheads to a certain degree. For instance, HEFT detects if running two tasks on the same processor would be faster than transferring the input data to another processor and executing them in parallel. However, HEFT does not take into account the long-term effects of such a decision. In the example, the inclination to avoid communication delays results in a schedule that leaves one processor idle.

2.2.2. Dynamic Workflow Scheduling

Dynamic workflow scheduling refers to making scheduling decisions during the execution of a workflow. In contrast, static scheduling methods assign tasks to compute resources before the execution of the workflow. The advantage of dynamic methods is their ability to react to changes in the computational infrastructure or incorporate information that becomes available during the execution of the workflow.

In principle, static scheduling methods can be turned into dynamic scheduling methods by repeatedly applying them at run time. To do so, one can repeatedly solve the partial scheduling problem obtained by removing the already completed tasks. However, static task graph scheduling heuristics typically expose computational complexity of at least $\mathcal{O}(v^2 p)$ where v is the number of tasks in the workflow, and p is the number of processing elements [Arabnejad and Barbosa, 2014]. Metaheuristics, like genetic algorithms, have even higher computational costs. Recomputing the schedule up to v times during the workflow execution might limit the number of tasks that can be scheduled, and thus processed, per time unit.

Due to speed requirements, dynamic and static scheduling methods typically differ in foresightedness. Static scheduling heuristics compute complete schedules, i. e., consider the furthest possible horizon and thus tend to be computationally more intensive. Dynamic scheduling heuristics typically make locally optimal choices. It is, of course, possible to design a static scheduling method that does not plan far ahead, e. g., by attempting to balance workload in round-robin fashion. Thus the degree to which a scheduling heuristic attempts to make locally optimal versus globally optimal decisions is a more distinctive criterion than the point in time at which scheduling decisions are made. However, due to the speed requirements described above, the two criteria, when decisions are made and how farsighted they are, typically correlate.

In addition to speed requirements, the reliability of information used in the scheduling process implies sensible horizons for planning. With unreliable information, e. g., highly uncertain task run times and data transfer times, making plans that extend far into the future is pointless. As described above, static scheduling heuristics can be turned into dynamic scheduling heuristics, but they would still require reliable knowledge of, e. g., task

run times and file transfer durations. In [Sonmez et al., 2010], seven dynamic scheduling heuristics are compared, motivated by the observation that the information required for static task graph scheduling is typically very unreliable when scheduling multiple workflows across multiple clusters. The heuristics are classified according to whether the planning information is assumed to be known, obtained at run time, or unknown. Planning information includes task run times, file transfer durations, and the current utilization of the compute cluster. The simplest compared method is the round-robin scheduler, which does not use any information for planning. Bux refers to such scheduling methods as knowledge-free scheduling methods [Bux, 2017]. The scheduling heuristic that requires the most information is a dynamic version of the HEFT algorithm that factors in the current utilization of different clusters measured at run time, which demonstrates the fluidity of the distinction between static and dynamic scheduling heuristics.

2.3. Batch Execution of Scientific Workflows

Due to the complexity of executing tasks on distributed compute infrastructures, a common approach to execute scientific workflows is to use a batch scheduler for executing tasks [Di Tommaso et al., 2017, Köster, 2014]. The workflow management system acts as an additional layer of software, automatically generating job submission commands for workflow tasks, which are then issued to the batch scheduler. However, various models for allocating cloud resources and batch scheduled resources exist. This section focuses on workflow execution on batch scheduled resources. For a broader introduction to the topic and specific techniques like glide-ins refer to [Juve, 2012, Schultz et al., 2017].

Section 2.3.1 provides an overview of batch schedulers and batch scheduler support of scientific workflow management systems. The implications of executing scientific workflows on batch-scheduled compute resources are discussed in Section 2.3.2. The resulting problem of picking the right amount of resources per task and a metric for assessing the quality of memory allocation decisions is discussed in Section 2.3.3.

2.3.1. Batch Schedulers

A batch scheduler is an application that simplifies the execution of programs across distributed computational resources. The execution of a program is referred to as a job. The interaction with the batch scheduler typically happens through command-line tools for allocating compute resources, submitting jobs to a queue for later execution, or monitoring running and pending jobs. Command-line interfaces allow a scientific workflow management to automate all interactions necessary for workflow execution. An overview of the batch schedulers supported by Pegasus, Nextflow, and Galaxy is given in Table 2.2.

A core feature of a batch scheduler is the coordination of multiple users' compute activities through arbitration of their competing resource requests, enforcing reservation limits on resource usage, job priorities, and user quotas. For instance, large scale compute clusters are often provisioned for the use across several research facilities, and access to compute resources is managed through a batch scheduler.

Distributed Resource Manager	Pegasus	Nextflow	Galaxy
Globus Resource Allocation Manager [Foster, 2006]	✓		
Moab HPC Suite [Adaptive Computing Enterprises, Inc., 2019a]	✓	✓	
HTCondor [Thain et al., 2005]	✓	✓	✓
Load Sharing Facility [IBM, 2019]	✓	✓	✓
Portable Batch System [Altair Engineering, Inc., 2019]	✓	✓	✓
Grid Engine [Univa Corporation, 2019]	✓	✓	✓
Slurm [Yoo et al., 2003]	✓	✓	✓
Torque [Adaptive Computing Enterprises, Inc., 2019b]	✓	✓	✓

Table 2.2.: Distributed resource managers supported by Pegasus, Nextflow, and Galaxy.

Resource Reservation

Each job has to specify resource limits prior to execution to control the amount of compute resources that users have access to, and the batch scheduler will enforce these limits by aborting jobs that exceed them. Typical parameters of resource reservations include the number of CPU cores, the number of compute nodes, main memory per core, disk storage, and the availability of accelerators such as GPUs or FPGAs. For clusters that have heterogeneous networks installed, some batch schedulers also allow specifying the interconnect that is used by a job, e. g., InfiniBand or Ethernet. Some batch schedulers, e. g., Slurm, allow resource requirement ranges, such as the minimum and maximum number of cores to allocate to a job. In addition to the amount and type of resources, a maximum duration for which access is granted to the resources usually needs to be specified. The Moab batch scheduler allows specifying a relation that defines allowed core allocations and the associated time limit, which not only provides more flexibility to the scheduler but also allows for more accurate job runtime estimations.

Support for different batch schedulers is necessary when having to execute scientific workflows across different compute sites, possibly managed by different organizations. As shown in Table 2.2, a considerable variety of batch scheduling software exists. This variety has led to the development of standards that provide a single interface for the specification of job parameters and resource requirements on a range of batch schedulers. An example is DRMAA, the Distributed Resource Management Application API [Tröger et al., 2016]. As an alternative to standards, meta-schedulers add another layer of software on top of batch schedulers and manually translate job specifications to the format supported by a specific batch scheduler. An example of a meta-scheduler is the Globus Resource Allocation Manager [Foster, 2006] and the Moab HPC Suite [Adaptive Computing Enterprises, Inc., 2019a]. Some scientific workflow management systems implement custom support for a range of batch schedulers [Di Tommaso et al., 2017] or provide extension mechanisms that allow users to add support for specific batch schedulers [Köster, 2014].

2.3.2. Batch Execution Model

Batch schedulers are complex pieces of software, comprising thousands of lines of codes. Thus, formal modeling and simulation of batch schedulers is challenging [Simakov et al., 2018]. This section states the assumptions made in this thesis about executing workflow tasks on a batch scheduler. These assumptions are used in Chapter 4 and Chapter 5 when simulating the execution of scientific workflows and assessing the resource efficiency achieved by learned resource usage predictions.

This thesis assumes that tasks have to specify in advance their memory usage limit, but it does not consider limits on execution time. Once granted, the allocated memory is available exclusively for the task that requested it. If the actual peak memory usage of a task exceeds its requested memory, it fails. The time that elapses until the task fails is referred to as time to failure.

Definition 2.19 (Task Attempt)**.** *A task in a workflow can be executed by submitting a corresponding job to a batch scheduler. A job submitted in order to execute a task is called an attempt to execute that task.*

Definition 2.20 (Time to Failure)**.** *If a task is allocated less memory than its peak memory usage, it fails. The wall-clock time elapsed between the start of the task and its termination is referred to as time to failure.*

In practice, the time to failure depends on the task's resource usage profile, i. e., at which point in time the memory usage first exceeds the assigned amount of memory. For instance, one could expect that when allocating drastically too little memory, the time to failure is shorter than in the case of allocating only slightly too little memory. However, this depends on the resource usage profile of the executed program. Generally, tasks failing early is the best case because it reduces the resources spent on a failed attempt. In the worst case, a task executes almost to completion before running out of memory. In the simulation experiments, the simplifying assumption of a fixed time to failure is made, i. e., the time to failure is the same regardless of the amount of allocated resources.

Resource Managers

Although the batch execution model is motivated by batch schedulers, it can be generalized to other distributed resource managers. A distributed resource manager is a piece of software that manages multi-user access to distributed compute resources. Batch schedulers fall into this category, but they are designed for the execution of arbitrary programs. A range of distributed resource managers exists that focuses on specific application models, such as the Apache Hadoop framework for task-based computing. In contrast to batch schedulers, which typically accept job submissions in the form of scripts, Hadoop jobs are submitted as Java programs that make use of the Hadoop application programming interface. This allows for more complex logic, such as dependencies between tasks. The Hi-WAY scientific workflow

management system [Bux, 2017] leverages Hadoop's task-based execution model for the execution of scientific workflows. Similar to executing a workflow using a batch scheduler, Hi-WAY has to specify the amount of CPU cores and memory when requesting resources for a task from Hadoop's resource manager YARN [Vavilapalli et al., 2013]. YARN also kills tasks that exceed their allocated memory; thus the batch execution model described here applies to scientific workflow execution on Hadoop as well.

2.3.3. Task Sizing

Under the batch execution model, users face a dilemma: trying to align resource requests to task resource usage closely increases the risk of task failures. Requesting more resources than the task needs reduces resource utilization. Another detrimental effect of requesting too much resources is that the wait time in a batch scheduler's queue also depends on the amount of requested resources, because small jobs can leverage resources left idle by other jobs. However, this effect is not considered in this thesis. As discussed in the previous section, the cost of task failures strongly depends on the time to failure. Optimizing the amount of allocated resources under the batch execution model is referred to as the task sizing problem [Tovar et al., 2018] and is addressed in detail in Chapter 5. This section introduces notation and metrics for the task sizing problem, as used in Chapter 4 and Chapter 5.

Tasks in scientific workflows frequently have vastly heterogeneous resource requirements [Juve et al., 2013, Rheinländer et al., 2016, Tyryshkina et al., 2019], and optimizing resource allocations is key to reduce workflow execution times. However, even experienced users may not be able to accurately estimate resource usage for their programs [Mu'alem and Feitelson, 2001]. As an alternative, estimates based on past performance measurements have been proposed [Witt et al., 2019a]. However, the resource usage of a task might strongly depend on the output of its ancestor tasks, which is reliably known only at runtime. In addition, data analysis workflows in a scientific context are "routinely unique" [Chang, 2015], meaning that past resource usage measurements on similar input data and compute infrastructure might not be available, contrary to repetitive production batch jobs in commercial scenarios [Alipourfard et al., 2017]. This motivates Chapter 4, which proposes an online learning approach to resource allocation.

To formalize what is assumed to be known about tasks prior to their execution and after their execution, the notion of a resource usage measurement set is introduced.

Definition 2.21 (Resource Usage Measurement Set)**.** *A resource usage measurement set* $D = (\tau, \tau^*, r, x)$ *specifies the run times* $\tau = \{\tau_1, \ldots, \tau_n\}$*, times to failure* $\tau^* = \{\tau_1^*, \ldots, \tau_n^*\}$*, peak memory usages* $r = \{r_1, \ldots, r_n\}$*, and input sizes* $x = \{x_1, \ldots, x_n\}$ *for* n *tasks.*

The run time denotes the wall clock time that elapses between the start and the completion of a task. The time to failure denotes the wall clock time that elapses between the start of a task and its termination in case of insufficiently allocated memory. The peak memory usage describes the maximum amount of memory used by the task throughout its execution. Finally, the input size x denotes the sum of the task's input file sizes. Input size is the

only quantity assumed to be known before the execution of a task. The other quantities are assumed to be measurable and known anterior to task execution. In practice, users tend to estimate resource requirements conservatively [Reiss et al., 2012] such that in production, only a small fraction of tasks may run out of memory and thus most τ_i^* are unknown. In this case, the simplifying assumption $\tau_i^* = \alpha\tau_i, 0 < \alpha \leq 1$ is made in Chapters 4 and 5.

Definition 2.21 implies that the run time and time to failure do not depend on the allocated resources. Hence resource heterogeneity is not accounted for by this model. Scheduling on heterogeneous resources is addressed in Chapter 3, whereas Chapter 4 and Chapter 5 assume homogeneous compute resources.

An approach to deal with task failures due to insufficient resources is to multiply the previous allocation by a factor $b > 1$ after each failure. This approach is referred to as exponential re-allocation. The case study in Chapter 5 provides an example of a workflow management system that applies this strategy in practice. This strategy is also applied in the simulation experiments in Chapter 4.

Definition 2.22 (Exponential Re-Allocation Function)**.** *Let u be an estimate of the peak memory usage of a task. An exponential re-allocation function f allocates u to the first task attempt and successively multiplies the allocation by a factor b in case of failures due to insufficient resources. The memory allocated to the j-th attempt is*

$$f(u, j) = b^{j-1}u \tag{2.5}$$

Memory Allocation Quality

To characterize the quality of memory allocation decisions, the memory allocation quality (MAQ) metric is used in this thesis. It is defined for a set of tasks and their allocation history, which reflects the allocation decisions for each task.

Definition 2.23 (Allocation history)**.** *An allocation history $A = (k, a)$ for a set of n tasks specifies the number of attempts $k = \{k_1, \ldots, k_n\}$ that have been made to execute the i-th task and how much memory $a = \{a_{11}, \ldots, a_{ik_1}, \ldots, a_{ij}, \ldots, a_{n1}, \ldots, a_{nk_n}\}$ was allocated to the j-th attempt of the i-th task.*

Memory allocation quality indicates the amount of wasted memory that results from underestimating or overestimating task memory usage. In contrast to average memory utilization, MAQ is not affected by the available memory of a compute infrastructure or task dependencies, which may force inevitable periods of low utilization.

Memory usage and wastage are measured in byte-seconds, i. e., the product of the amount of allocated memory and the duration of the allocation. For the quantification of used and wasted memory, two cases are distinguished. In the case of task failure, all of the memory

allocated to this task attempt is considered wasted. In the case of task success, only the difference between allocated memory and the task's peak memory usage is considered wasted. The amount of usefully allocated memory corresponds to the product of task run time and task peak memory usage.

Definition 2.24 (Wasted Memory)**.** *Let D be a resource usage measurement set and A an allocation history for the tasks in D. The Memory wastage of the j-th attempt of the i-th task corresponds to the product of excess allocation and the attempt's run time. On successful attempts, the excess allocation is the difference between allocated resources a_{ij} and peak usage r_i. On failed attempts, the complete allocation is considered as excess allocation.*

$$\mathit{wastage}(\tau_i, \tau_i^*, r_i, a_{ij}) = \begin{cases} (a_{ij} - r_i)\tau_i & \text{if } a_{ij} \geq r_i \\ a_{ij}\tau_i^* & \text{otherwise} \end{cases} \tag{2.6}$$

Definition 2.25 (Used Memory)**.** *Let D be a resource usage measurement set and A an allocation history for the tasks in D. The memory usage of a successful task attempt corresponds to the product of its run time and its peak memory usage. On failed attempts, none of the allocated memory counts as usefully allocated.*

$$\mathit{usage}(\tau_i, \tau_i^*, r_i, a_{ij}) = \begin{cases} r_i\tau_i & \text{if } a_{ij} \geq r_i \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

Memory allocation quality is defined as the used byte-seconds U divided by total allocated byte-seconds $U + W$. Accordingly, a MAQ of 100% corresponds to no wastage.

Definition 2.26 (Memory Allocation Quality)**.** *Let D be a resource usage measurement set. Memory allocation quality describes the fraction of usefully allocated memory for an allocation history $A = (k, a)$.*

$$U = \sum_{i=1}^{n} \sum_{j=1}^{k_i} \mathit{usage}(\tau_i, \tau_i^*, r_i, a_{ij}) \tag{2.8}$$

$$W = \sum_{i=1}^{n} \sum_{j=1}^{k_i} \mathit{wastage}(\tau_i, \tau_i^*, r_i, a_{ij}) \tag{2.9}$$

$$\mathit{MAQ} = \frac{U}{U + W} \tag{2.10}$$

2.4. Memory Usage Prediction

This section gives an overview of state-of-the-art memory usage prediction by comparing nine approaches, including the one presented in Chapter 5. In the past years, the interest in memory usage prediction has increased, motivated by a desire to automate and improve resource allocation in systems that require resource reservations. This research is relevant for executing workflows using the batch execution model described in Section 2.3.2.

This section is structured as follows. First, variants of the memory usage prediction task are defined. Second, an overview of common features in the training data is given. Third, an overview of the machine learning methods used in the reviewed research literature is given. Fourth, evaluation metrics are reviewed, and specifics of the resource usage prediction problem are discussed. Table 2.4 uses the introduced definitions to characterize the reviewed literature. Table 2.5 provides an overview of the data sets used in the papers. A more comprehensive review of predictive performance modeling can be found in [Witt et al., 2019a], which also covers the prediction of execution times, data transfer times, and queue times.

Prediction Tasks

Several memory-related prediction tasks exist, e. g., forecasting the memory usage over time [Schmidt et al., 2018], predicting the performance impact of contention for on-chip memory resources such as bandwidth and cache space [Chatzopoulos et al., 2016, Zhao et al., 2016, Govindan et al., 2011], or predicting cache access patterns [Marin and Mellor-Crummey, 2004], This overview focuses on predicting the maximum amount of memory used during the execution of a program. In the reviewed literature, four variations of the task have been identified:

- Regression (R). Predict the peak memory usage during the execution of a program. This is the most commonly addressed task. It can be used to automate memory allocation or to provide feedback to users on whether their estimates are sufficient.
- Interval prediction (I). Predict an interval that contains the peak memory usage. Among the reviewed papers, only [Tyryshkina et al., 2019] addresses this task. A possible use case is to predict confidence intervals to allow schedulers balancing resource efficiency and failure probability.
- Regression by classification (C). Predict which of a predefined set of intervals contains the peak memory usage. Often, coarse-grained predictions are sufficient. In some cases, users are only interested in whether a job consumes little or much memory. Jobs with large memory requirements then receive special treatment, e. g., by submitting them to dedicated job queues or executing them in different virtual machine types.
- Active learning (AL). Select jobs to execute to build a training set for either of the above models while minimizing the total execution cost of collecting the training instances. Benchmarking the memory requirements of programs can be time-consuming. Intelligently selecting examples may help in reducing the overhead of collecting training data.

Features

The different approaches base their predictions on various information. The surveyed papers do not use a consistent way of grouping features. Here, features are broadly classified in features related to *what* is computed, features related to *how* the computation is carried out, and features that implicitly characterize similarities between jobs.

The first class of features indicates logical aspects of computation, i. e., what computation is performed. These features describe the program that is executed and its input data. For instance, a name or identifier of the program, the program version, and program parameters are frequently used to predict peak memory consumption. Program parameters may comprise problem-specific parameters, e. g., describing a physical phenomenon to be simulated, or methodical parameters, e. g., the resolution of the simulation. Occasionally, descriptions of program purpose, e. g., sequence alignment, or the project phase, e. g., design phase or implementation phase, are available [Taghavi et al., 2016].

Inputs can be characterized, for instance, through file sizes, data set identifiers, or occasionally logical descriptions of file contents, e. g., the number of amino acids in a sequence, which may be non-linearly related to compressed file size.

The second class of features is here referred to as features related to the physical aspect of execution. For instance, there are typically multiple semantically equivalent ways to carry out an execution, e. g., the partition size in a parallel problem should not affect the outcome. CPU and memory requirements are sometimes correlated. Several approaches thus take into account requested non-memory resources, such as the number of nodes, cores, and time requested for the job. In other cases, a program takes care of auxiliary steps such as the decompression of an input file. Whether the necessity of such steps and the resulting variations in resource usage could be predicted by using file extension as a feature [Tyryshkina et al., 2019].

Logical and physical aspects of computation can also be indirectly inferred from submission properties such as the user who submitted a job, the project the job belongs to, and the submission time. While these features seem only loosely related to what or how is computed, they can expose similarities between jobs that allow for re-use of previous measurements. For instance, a user may work on a particular problem with similar resource requirements over a certain time, or organizational processes cause specific jobs to be issued on certain days of the week. Therefore, several models incorporate a user's identity, group, or role in an organization. Harnessing implicit similarities or recent history can thus improve predictions of resource usage. Historical data can, for instance, be aggregated by computing average resource usage per user or project and using it as a feature [Andresen et al., 2018].

Methods

The surveyed approaches employ a range of machine learning methods. The abbreviations in Table 2.4 denote the following machine learning approaches:

- Linear regression (LR). If all features are numerical and the relationship between memory usage and features is approximately linear, linear regression can be used. However,

ordinary least squares regression provides failure rates around 50%. Variations of the approach thus feature alternative loss functions [Tovar et al., 2018, Witt et al., 2019b].

- Regression tree (RT). As an additional layer on top of a regression model, decision trees can be used to select a regression model conditional on the input. Standard methods include CART [Breiman et al., 1984], MARS [Friedman, 1991], CHAID [Miner et al., 2009], and PQR [Gupta et al., 2008].
- Neural network (NN). Like regression trees, neural networks are capable of dealing with non-linear relationships. Out of the large canon of neural network architectures, multilayer perceptrons [Hastie et al., 2009] are a simple approach to regression tasks.
- Support vector machine (SVM). Support vectors are another approach to construct decision boundaries and regression models [Cristianini and Shawe-Taylor, 2009].
- K-nearest neighbors (KNN). A baseline approach for machine learning is to define a similarity measure and perform classification or regression by aggregating the classes or outcomes of the nearest neighbors of an unknown instance [Atkeson et al., 1997].
- Gaussian processes (GP). Gaussian processes provide estimates of uncertainty in addition to predictions, based on the dispersion of the training data in the vicinity of an input [Rasmussen and Williams, 2006].
- Time series (TS). When the training data can be ordered over time, e. g., by looking at the recent job history of a user, time series methods may be used to predict the peak memory usage of the next job. Standard approaches include autoregressive-moving-average (ARIMA) and Kalman filters (KF) [Box et al., 2015].

Evaluation Criteria

The criteria used for evaluation are summarized in Table 2.3. In addition to standard machine learning metrics like RMSE, MAPE, and R^2, some metrics have been used that are particularly relevant to the task of predicting peak memory usage. These metrics take into account the asymmetric costs of overprediction and underprediction. The most basic metric is the failure rate, i. e., the fraction of predictions below the actual peak memory usage. Note that failure rate may not be a comprehensive characterization of the number of task failures in the system because most of the publications do not take into account that failed tasks have to be restarted with a new prediction. When predictions are successively increased upon task failure, e. g., by doubling the allocation, the number of failures depends on how strongly memory usage is underestimated. This can be quantified using average underprediction (AUP). However, reducing underprediction usually increases average overprediction (AOP), which thus should also be reported. Finally, memory allocation quality (Definition 2.26) can be used to express the fraction of used resources relative to allocated resources.

Notation	Metric	Definition	Task
AIC	Akaike information criterion	$2k - \ln \hat{L}$	All
RMSE	Root mean squared error	$\sqrt{\sum (f_i - y_i)^2 / n}$	R
R^2	Coefficient of determination	$1 - \frac{\sum (f_i - y_i)^2}{\sum (f_i - \bar{y})^2}$	R
MAPE	Mean absolute percentage error	$1/n \sum \lvert (f_i - y_i)/y_i \rvert$	R
FR	Failure rate	$\lvert \{f_i \mid y_i < f_i\} \rvert / n$	R, I, C
AUP	Average under-prediction	$1/n / \mathsf{FR} \sum_{f_i < y_i} y_i - f_i$	R
AOP	Average over-prediction	$1/n / (1 - \mathsf{FR}) \sum_{f_i > y_i} f_i - y_i$	R
ACC	Accuracy	$\lvert \{f_i \mid y_i \in f_i\} \rvert / n$	C, I

Table 2.3.: Evaluation metrics. f_i denotes the predicted peak memory usage and y_i denotes the actual peak memory usage of the i-th of n tasks in the evaluation data set. $\bar{y}$ denotes the average memory usage, $\hat{L}$ denotes the maximum likelihood of a model, and k denotes the number of model parameters.

Reference	Features	Methods	Task	Evaluation
Chapter 5, [Witt et al., 2019b]	Input size, data set id, program id	LR (asymmetric loss)	R	AOP, AUP, MAQ (≈75%)
[Li et al., 2019]	Program, job, and submission features, user estimates	Random forest	C	ACC (≈90-95%)
			R	MAPE (≈10-35%)
[Tyryshkina et al., 2019]	Program id, version, and parameters, data set id, file size and extension	Random forest	R	R^2 (≈60-85%)
			I	ACC (≈90%)
[Tovar et al., 2018]	Program id or category	LR (intercept only, asymmetric loss)	R	FR, MAQ (≈70%)
[Andresen et al., 2018]	User id, role, and estimates, user statistics	LR, RT (CART)	R	R^2 (≈15-30%)
[Duplyakin et al., 2018]	Number of nodes, method parameters, problem parameters	Bayesian optimization, GP	AL	RMSE convergence speed, total costs
[Rodrigues et al., 2017]	User, group, and queue, submission time, user estimate	Ensemble (SVM, RF, NN, KNN), sliding window	C	ACC (≈75-90%)
[Taghavi et al., 2016]	User statistics and estimates, project, requested processors, job type, recent job resource usage	NN, TS (ARIMA, KF), RT (CART, MARS, CHAID)	R	R^2 (≈93%), RMSE, AIC, AOP, AUP
[Matsunaga and Fortes, 2010]	Input description, hardware characterization	KNN, LR, RT (PQR)	R	MAPE (≈1-5%)

Table 2.4.: Overview of state-of-the-art approaches for predicting memory usage.

Reference	Dataset	Jobs	Period	System	Open
Chapter 5, [Witt et al., 2019b]	Physics simulation workflows from the IceCube project	727×10^3	10 months	Multiple	Yes
[Li et al., 2019]	Jobs from IBM customers	10^6	40 days	LSF	No
[Tyryshkina et al., 2019]	Bioinformatics jobs on `usegalaxy.org`	10^4	1 month	SLURM	Yes
[Tovar et al., 2018]	Jobs from three workflows from the physics and bioinformatics domain	500×10^3	1 month	HTCondor	Yes
[Andresen et al., 2018]	Job traces from campus cluster	*NA*	8 years	SGE	No
[Duplyakin et al., 2018]	Adaptive mesh refinement for shock bubble simulation	600	*NA*	SLURM	Yes
[Rodrigues et al., 2017]	IBM research cluster and a production system	25×10^3	*NA*	LSF	No
[Taghavi et al., 2016]	Chip design jobs, e. g., floor planning	500×10^3	1 month	LSF	No
[Matsunaga and Fortes, 2010]	Bioinformatics program executions	7×10^3	*NA*	*NA*	Yes

Table 2.5.: Overview of the data used in the memory prediction papers. "Jobs" refers to the approximate number of instances in the training data. "Period" refers to the time period during which the data was collected. "System" refers to the batch scheduler used to run jobs. "Open" refers to whether the data was published or can be obtained from the authors.

3. Level-Order Sampling for Static Task Graph Scheduling

This chapter presents a novel algorithm for the static task graph scheduling problem, as introduced in Section 2.2.1. A well-cited heuristic for this problem is the Heterogeneous Earliest Finish Time algorithm (see Section 2.2.1), which comprises two components. The first component is a heuristic that assigns priorities to tasks. The second component is a processor assignment method that chooses the processor, which minimizes the finish time for a given task. It has been shown that replacing the first component by a different weighting scheme can yield better solutions, although no single weighting scheme always performs best [Zhao and Sakellariou, 2004]. This is due to the many degrees of freedom in the scheduling problem, which makes it easy to construct workflows that exploit the weaknesses of certain weighting schemes. A natural approach to achieve good performance over a wide range of situations is thus to generate more rankings.

In this chapter, a randomized search for rankings is proposed that capitalizes on HEFT's processor assignment method but replaces its task prioritization heuristic. The proposed Level-Order Sampling (LOS) method explores alternative rankings by randomly modifying task priorities. LOS partitions the graph into levels, such that randomly permuting task priorities of a level maintains a topological ordering of the tasks (see Section 2.1.2). The decision which partition to modify next is based on the improvements gained from modifying different levels. LOS also considers long-term dependencies between scheduling decisions by repeatedly evaluating schedule variations affecting different portions of the task graph.

A time-budgeted method is proposed to offer users more flexibility in balancing the conflicting goals of scheduling quality and speed. LOS yields up to 40% shorter schedules than HEFT and often outperforms weighting schemes from the literature on a set of 4 500 random task graphs.

The chapter is structured as follows. Section 3.1 presents the LOS algorithm for randomized scheduling. Section 3.2 presents the results of the experimental evaluation. Section 3.3 reviews related work specific to static task graph scheduling. Section 3.4 summarizes the results and puts them into perspective for this thesis.

3.1. Method

This section introduces the Level-Order Sampling (LOS) method for static task graph scheduling. LOS performs two kinds of steps that are repeated in an alternating fashion, as shown in Figure 3.1. An exploitation step consists of generating random variations of a reference solution, as outlined in Figure 3.2. An exploration step consists of replacing the current

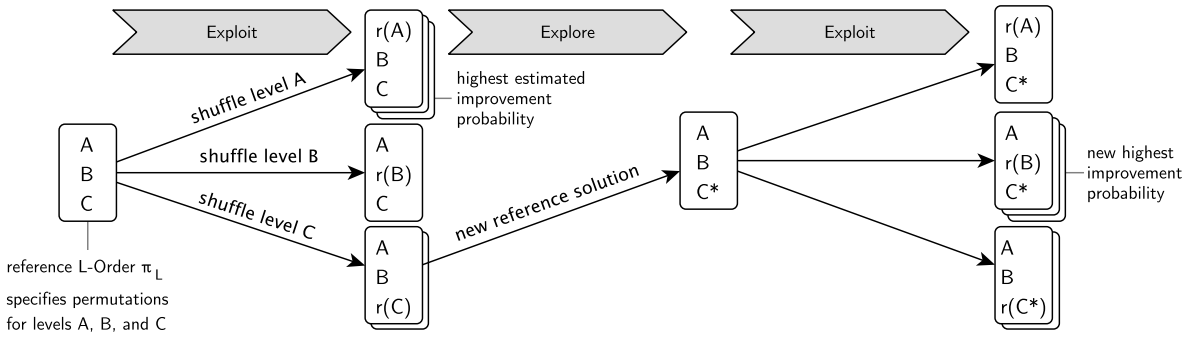

Figure 3.1.: Overview of the LOS method and its interplay between exploitation and exploration. Exploitation refers to generating multiple variations of a reference solution by shuffling individual levels, denoted by $r(\cdot)$. LOS prefers those levels for shuffling that have a high estimated probability of finding a better solution. Exploration refers to picking the best variation as a new reference to exploit.

reference solution with an improved solution. This improvement process is repeated until a time budget is exhausted. The complex part lies in the method used to find improvements of a reference schedule, which is shown in Figure 3.2. The structure of the chapter follows the steps in the flow chart:

- Section 3.1.1 introduces level-based partitioning, L-Orders, and a shuffle operator.
- Section 3.1.2 introduces a way to estimate the improvement probability of a level, i. e., the probability of reducing the current makespan by shuffling the level.
- Section 3.1.3 shows how to select levels for shuffling based on their improvement probabilities and how to estimate the wall clock time until the next improvement.
- Section 3.1.4 introduces an approach to balance the search for more improvements and the search for bigger improvements, which yields the final Level-Order Sampling method.

3.1.1. L-Order Sampling

At the core of LOS is the idea of using an ordering of the tasks to guide the construction of the schedule (see list scheduling heuristics Section 2.2.1). The idea is to profit from HEFT's proven processor assignment method but using different orderings of tasks. HEFT's processor assignment algorithm requires that the ordering of the tasks is a topological one. This section presents L-Orders as a simple and efficient approach to randomly sample topological task orderings. The proposed scheme also provides a way to group variations of a task ordering, which allows for a focused search for improved orderings.

Recall from Section 2.1.2 that the level of a task corresponds to the longest path to an exit task. Recall from Section 2.2.1 that an order is a permutation of the tasks in a workflow.

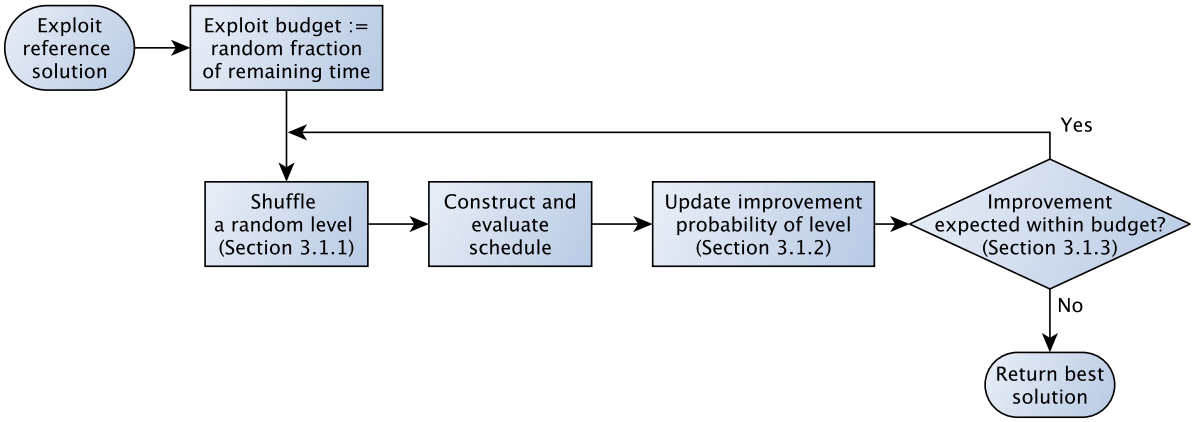

Figure 3.2.: During an exploit step, random variations of a reference schedule are generated by shuffling random levels of the L-Order that corresponds to the reference schedule.

An L-Order is a sorting of the tasks with respect to their level. Figure 3.3 exemplifies levels and orders.

Definition 3.1 (L-Order)**.** *Let $G = (V, E, w, c)$ be a communication DAG and π be a topological ordering of the tasks V. An ordering π_L is an L-Order if and only if the levels of the tasks are non-increasing with respect to their ranks:*

$$\forall T_i, T_j \in T \colon \pi_L(T_i) > \pi_L(T_j) \Rightarrow \mathit{level}(T_i) \geq \mathit{level}(T_j) \tag{3.1}$$

Lemma 1. *Every L-Order is a topological order. Otherwise, a pair of tasks with $\pi_L(T_i) > \pi_L(T_j)$ could exist such that $T_j \prec T_i$. However, $\mathit{level}(T_j) = 1 + \max_{T_j \prec T_i} \mathit{level}(T_i)$ implies $\mathit{level}(T_j) > \mathit{level}(T_i)$, which contradicts the assumption. On the other hand, a topological order is not necessarily an L-Order, as shown in Figure 3.3.*

L-Orders have the convenient property that random variations can be obtained by randomly permuting the tasks with the same level while maintaining the topological ordering. This is called shuffling.

Definition 3.2 (Shuffle Operation)**.** *To shuffle the k-th level of an L-Order, replace the order of the nodes of level k with a random order of these nodes. More formally, pick a random bijection $r \colon R \to R$ on the set of the ranks R of the T_i with level k and let the new L-Order π'_L be:*

$$\pi'_L(T_i) = \begin{cases} r(\pi_L(T_i)) & \text{if } \mathit{level}(T_i) = k \\ \pi_L(T_i) & \text{otherwise} \end{cases} \tag{3.2}$$

Shuffling a level requires $\mathcal{O}(k)$ time, where k is the number of modified positions in the list, i. e., the number of tasks with the according level. In comparison, computing a random

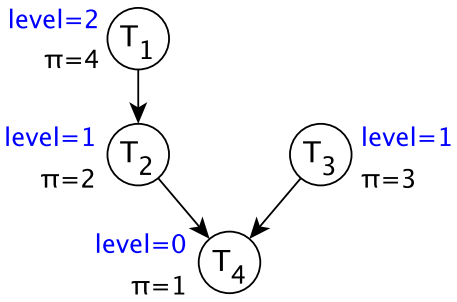

Figure 3.3.: Example dag, its task levels, and an ordering π. The edges of the dag give $T_1 \prec T_2, T_1 \prec T_4, T_2 \prec T_4, T_3 \prec T_4$. Since neither $T_2 \prec T_3$ nor $T_3 \prec T_2$, both tasks can be executed in parallel. The dag has three topological orders: (T_1, T_2, T_3, T_4), (T_1, T_3, T_2, T_4), (T_3, T_1, T_2, T_4), the first two of which are L-Orders. © 2018 IEEE

topological order requires $\mathcal{O}(n)$ time, where n is the number of tasks in the dag. The set of L-Orders is closed under the operation of shuffling since changing the order of tasks with equal levels maintains the requirement of non-increasing task levels. Variations can be naturally grouped by the level that was shuffled to produce them.

Definition 3.3 (Region)**.** *Let π_L be an L-Order. A region refers to the set of all possible L-Orders resulting from shuffling a specific level of π_L.*

A solution to the scheduling problem can be obtained by applying HEFT's processor assignment method (see Section 2.2.1) to an L-Order. The quality of the L-Order is defined as the quality of the resulting schedule, which is here measured as its makespan (also see Section 2.2.1). In the following, the terms L-Order and solution will be used interchangeably. Supplemental Figure B.1 shows an example of the impact of shuffling on the solution quality, demonstrating that randomized task priorities can yield substantially improve over HEFT.

3.1.2. Improvement Probability

This section describes how to estimate the improvement probability, i. e., the probability of finding a solution with a makespan better than a given reference makespan r in a certain region. This is used to select regions to sample solutions from for comparison with the current best solution (Section 3.1.3).

Let π_L be an L-Order and k the level that defines the region for which to estimate the improvement probability. The basic idea is to analyze the distribution of makespans obtained when repeatedly sampling L-Orders from that region. It is useful that the makespans are often approximately normal distributed (see supplemental Figure B.1 for an example makespan distribution), but the proposed method does not rely on this assumption. If, however, the makespans are observed to be approximately normally distributed, the cumulative distribution function of the corresponding normal distribution can be used to estimate improvement

probabilities. Intuitively, fitting a normal allows extracting more information from the sampled makespans than considering only the fraction of makespans below the current best solution.

Normal Distributions

Let μ_k denote the average makespan of the solutions obtained so far by shuffling level k. Let σ_k denote the upper endpoint of a 95% confidence interval for the standard deviation of the makespans. Using the upper endpoint of the interval to estimate the standard deviation results in optimistic estimates of improvement probabilities when few samples are available because more variance increases the chances to find better solutions. This increases the attractiveness of the region for sampling, as explained in the next section. If the normal distribution provides a good fit to the distribution of makespans, the probability to randomly select an L-Order with makespan m at least as good as a reference makespan r by shuffling level k can be estimated using the cumulative distribution $F_{\mathcal{N}}$ of a normal distribution with the according parameters:

$$p^k_{\mathcal{N}}(m \leq r) = \frac{1}{2} F_{\mathcal{N}}(r; \mu_k, \sigma^2_k) \tag{3.3}$$

Non-Normal Distributions

After each newly sampled solution, the number of expected improvements is compared to the number of found improvements to test whether improvement probability can be approximated with a normal distribution. Let n_k denote the number of solutions sampled from the region corresponding to level k. The number $\hat{n}_k$ of solutions better than r is compared to the number of better solutions that one would expect to see, under the assumption that $p^k_{\mathcal{N}}(m \leq r)$ follows Equation 3.3. Sampling the makespan distribution is modeled as a Bernoulli trial that succeeds with $p^k_{\mathcal{N}}(m \leq r)$. Using the cumulative distribution function F_B of a Binomial distribution to obtain the probability of observing at most $\hat{n}_k$ successes among n_k trials, the normality hypothesis is rejected if an improbably ($p < 5\%$) small number of improvements is found in a region. It is then assumed that Equation 3.3 does not apply for the current region. In this case, the simpler empirical probability $p^k_r(m \leq r) = \frac{\hat{n}_k}{n_k}$ is used instead. To avoid zero improvement probabilities, $\hat{n}_k$ is set to one as long as no improvement has been found.

In summary, the improvement probability is based on the cumulative distribution function of a fitted normal when improvement expectations have not been significantly violated, and on a simple success rate otherwise.

$$p^k(m \leq r) = \begin{cases} p^k_r(m \leq r) & \text{if } F_B(\hat{n}_k; n_k, p^k_{\mathcal{N}}(m \leq r)) < 5\% \\ p^k_{\mathcal{N}}(m \leq r) & \text{otherwise} \end{cases} \tag{3.4}$$

3.1.3. Exploiting Improvement Probabilities Across Regions

This section describes a process for searching improvements of a given reference L-Order π_L, based on the improvement probabilities described in the previous section. This process is referred to as an exploitation phase (see Figure 3.1).

The idea is to repeatedly and randomly select a level l to shuffle for obtaining a new solution, favoring levels with high improvement probability. After evaluating the new solution by computing the makespan of the resulting schedule, the level's improvement probability is re-estimated. Let $p(l = k)$ denote the *level selection probability*, the probability to pick level k to generate the next variation of the reference order. The level selection probability is defined as the level's share of the improvement probabilities summed over all levels. Using the estimated improvement probabilities for the different levels as defined in Equation 3.4, the level selection probability for a specific level k is defined as:

$$p(l = k) = \frac{p^k(m \leq r)}{\sum_i p^i(m \leq r)} \tag{3.5}$$

Probabilistic Level Selection

Probabilistic level selection ensures that most of the compute time is spent on promising levels, but also allows to sample from less promising levels from time to time. The motivation for this is that improvement probabilities are only estimated, and randomization gives less promising levels a chance to reveal improved solutions. The use of the upper endpoint of a confidence interval in the estimation of the improvement probabilities encourages sampling from regions with few observations, which is desirable to obtain more reliable estimates. For levels that have less than two evaluated solutions, $p^k_{\mathcal{N}}$ is set to one. The level selection process is illustrated in Figure 3.4. Supplemental Figure B.2 shows an example of the evolution of level selection probabilities as more solutions are sampled. The figure shows how initially promising levels can become less promising as additional solutions are evaluated, and vice versa.

For levels with few tasks, permutations are not obtained through shuffling but enumerated and randomly drawn without replacement. After having evaluated all of a level's permutations, its improvement probability is set to zero. Enumeration avoids evaluating the same solution twice. For levels with more tasks, the number of permutations grows so fast that the probability of repeatedly sampling the same solution quickly approaches zero.

Time to Improvement

The expected number of samples until the next improvement can be estimated based on the estimated improvement probabilities, Let L be the number of levels with $p^k(m \leq r) > 0$. The required number of additional samples until a better solution is found in a given level is estimated as $1/p^k(m \leq r)$ multiplied by the level's selection probability $p(l = k)$. The overall expected number of additional samples until improvement s_{improv} results from summing over

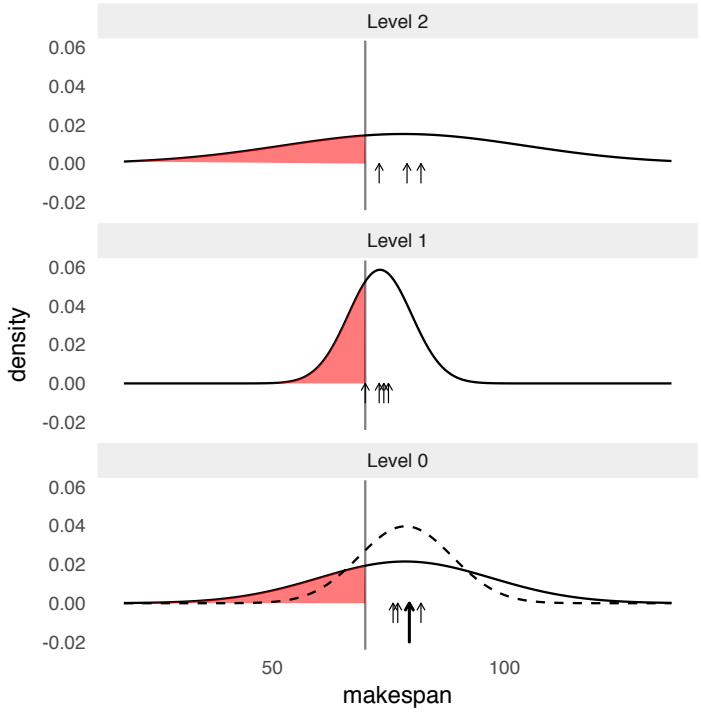

Figure 3.4.: The level selection process uses the estimated improvement probabilities (red area) to favor promising levels for sampling new variations. The makespans of the L-Orders sampled so far (black arrows on the x-axis) are used to fit a normal distribution for each level. Level 0 has the highest improvement probability, which is reduced by the new solution (bold arrow, dashed distribution). In the next step, level 1 has the highest estimated improvement probability. Note that standard deviations are based on 95% confidence intervals, such that the fitted distributions are much wider when few solutions are available, e. g., for level 2.

all levels.

$$s_{\text{improv}}(r) = \frac{L}{\sum_i p^i(m \leq r)} \tag{3.6}$$

Given an estimated compute time t per solution sample, the wall clock time until improvement t_{improv} is calculated as $t \cdot s_{\text{improv}}(r)$. In practice, the average of all measured solution evaluation times so far is used as the estimated compute time.

3.1.4. Balancing Quality and Quantity of Improvements

This section combines the previous components into a time-budgeted, randomized scheduling algorithm. The overall LOS algorithm takes a communication dag to schedule and a point in time until scheduling shall finish. It repeatedly applies the exploitation method described in the previous section to find improvements to a current solution. After each exploitation phase, the best solution replaces the current reference solution, enabling interacting modifications on different levels of the directed acyclic graph. This section describes LOS's approach to balancing the quality and quantity of improvements, i. e., the length and number of exploitation phases.

Randomized Alternation

Long exploitation phases favor exploitation, i. e., finding larger improvements to a reference order; short exploitation phases favor a large number of smaller improvements. Preliminary experiments have shown that sometimes longer exploitation phases are favorable, while in other cases shorter exploitation phases give better results. Randomization was found to perform well on the task of automatically selecting the number and length of exploitation phases. The length of an exploitation phase is chosen uniformly at random between 5% and 50% of the remaining time budget. The downside is that introducing more sources of randomness potentially also increases the variance of the performance of the method, which, however, was found to be sufficiently low during evaluation (see Section 3.2.3). An exploitation phase is prematurely aborted when the expected time to improvement (see Section 3.1.3) exceeds the budget of the exploitation phase. The idea is to adaptively shorten exploitation phases in support of leaving dead ends in the search space through a series of small changes. The overall design of an exploitation phase is summarized in Algorithm 1.

The alternation of exploitation and exploration phases is shown in Algorithm 2. Note that alternating exploration and exploitation is essential because only one level is modified relative to the reference order during each exploitation phase. Using an improved solution as new reference allows for combining modifications of task priorities in different parts of the workflow.

3.2. Experimental Results

This section summarizes the experimental setup, the baseline methods, and the performance and behavior of LOS.

Algorithm: Exploitation Phase

Method *Exploit(initial solution π_L)*:

best order $\pi_L^* \leftarrow \pi_L$;
best makespan $r \leftarrow$ makespan of π_L;
`// Select length uniformly at random`
partial budget $\leftarrow$ remaining time $\cdot\, \mathcal{U}(0.05, 0.5)$;
deadline $\leftarrow$ current time+partial budget;
`// Expected wall-clock time until an improvement is found`
$t_{\text{improv}} \leftarrow 0$;
while *current time*$+t_{improv} <$ *deadline* **do**
  $p(l = k) = \frac{p^k(m \leq r)}{\sum_i p^i(m \leq r)}$;
  randomly select a level l with probability $p(l = k)$;
  $\pi'_L \leftarrow \pi_L^*$ with level l shuffled;
  `// Evaluate makespan of new solution`
  $m' \leftarrow$evaluate(π'_L);
  add makespan m' of π'_L to makespan distribution of level l;
  `// New solution is better`
  if $m' < r$ **then**
    $r \leftarrow m'$;
    $\pi_L^* \leftarrow \pi'_L$;
    `// New reference makespan invalidates current estimates`
    update all improvement probabilities $p^k(m \leq r)$;
  end
  else
    `// Only data has changed, not reference makespan` r
    update improvement probability $p^l(m \leq r)$ of level l;
  end
  update t_{improv} according to new improvement probabilities $p^k(m \leq r)$;
end
return π_L^*
end

Algorithm 1: The exploitation algorithm samples new L-Orders from the most promising levels as long as the expected time to improvement is below a fraction of the remaining time budget.

```
Algorithm: Level Order Sampling
Method Schedule(communication dag G, deadline):
    solution ← random L-Order of G;
    while current time < deadline do
        //exploit
        best improvement ← Exploit(solution);
        //explore
        solution ← best improvement;
    end
    return solution;
end
```

Algorithm 2: The level order sampling algorithm combines exploration and exploitation into a budgeted search for an L-Order that minimizes makespan. The amount of time spent in the exploit subroutine is chosen randomly (see Section 3.1.3).

3.2.1. Input Instances

For evaluation, random dags are generated using the method described in [Krapivsky and Redner, 2001]. The compute time $w(T_i, p_j)$ of a task on a processor is sampled uniformly at random in range $[1, 100]$. The edge weights $w(T_i, T_j)$ are also sampled uniformly in range $[1, 100]$. The data transfer rates of the network links are set to one. Dags of size $n \in \{32, 64, 128, 256, 512\}$ are generated and scheduled to $p \in \{3, 10, 30\}$ processors. The expected number of levels in the random graphs depends on the size of the graphs. In a sample of 1000 random graphs of size 100, there were 7.3 ± 1.1 levels on average, with a minimum of five levels and a maximum of eleven levels.

The combination of a dag and a number of processors is referred to as an *input instance*. For each input instance, a schedule is computed using the original HEFT algorithm, and its makespan is used as a reference to assess the quality of the schedules delivered by the baseline method and LOS. Since LOS relies on randomization and thus exhibits variance in performance, LOS is run three times on each input instance, and each run is referred to as an *experiment*.

In total, 4500 random dags have been generated, 900 of each size. Each of the dags was combined with each number of processors, giving 13 500 input instances. Using three repetitions, this results in 40 500 experiments. The time budget of LOS has been set to 30 seconds for dags of size 32 and 64, and to five minutes for dags of size 128, 256, and 512. In each experiment, four instances of the LOS algorithm are run in parallel, and the best solution is chosen. Overall, 426 072 399 schedules have been considered during the experiments by LOS.

3.2.2. Baseline Ranking Methods

LOS is compared to two baseline methods, the original HEFT method [Topcuoglu et al., 2002] and the ranking methods proposed by Zhao et al. [Zhao and Sakellariou, 2004]. Zhao

et al. use an ensemble of twelve methods to replace HEFT's task weighting method. The different methods are variations of HEFT's approach, which first averages computation and communication costs and then applies the upward rank formula (see Section 2.2.1). Instead of averaging, Zhao et al. evaluated aggregation by taking the median, maximum, and minimum of the computation costs across processors. Likewise, replacing average communication costs by median, maximum, minimum, computation-specific maximum, and computation-specific minimum communication costs have been considered. The computation-specific minimum communication cost between two tasks is the communication cost between the two processors that minimize the computation times of the tasks[1]. The computation-specific maximum communication time is defined analogously. Instead of using HEFT's upward rank, the above aggregation methods have also been applied to compute downward ranks.

Definition 3.4 (Downward Rank)**.** *Let $W = (V, E, w, c)$ be a communication dag and $\bar{w}$ and $\bar{c}$ aggregated computation and communication costs, respectively. The downward rank of a task is defined recursively as*

$$\mathit{drank}(T_j) = \begin{cases} 0 & T_j \text{ is an entry task} \\ \max\{\bar{w}_i + \bar{c}\,(T_i, T_j) + \mathit{drank}(T_i) \mid (T_i, T_j) \in E\} & \text{otherwise} \end{cases} \tag{3.7}$$

To evaluate the approach of Zhao et al., the ordering that produces the schedule with the shortest makespan is selected. Since the ensemble contains HEFT's original approach, i. e., averaging with upward rank, this approach never performs worse than HEFT. The same fallback mechanism has been applied to LOS by returning HEFT's solution in case LOS has not found a better solution.

3.2.3. Makespan Reductions

The improvements of LOS over HEFT and Zhao's alternative ranking methods are computed on different dag sizes and numbers of processors. For each input instance, the relative makespan is computed as the ratio between the makespan of the best schedule delivered by a method (Zhao, LOS) and the makespan of HEFT's schedule. Figure 3.5 shows the distributions of the relative makespans. LOS clearly outperforms the baseline method, giving shorter median makespans and longer tails towards short makespans in most of the cases. A striking effect is that both the baseline and LOS deliver diminishing improvements for larger dags. An interesting exception are the graphs of size 512 on 30 processors, which suggest that LOS could perform well on even larger input instances. However, the relationship between dag size and number of processors is complex; for example, the baseline is better on dags with 256 tasks on ten processors, which gives a similar task-to-processor ratio as the previous example.

[1] The paper is not clear about ambiguous cases, since different pairs of processors may exist that minimize the computation times of the tasks and yet have different transfer rates.

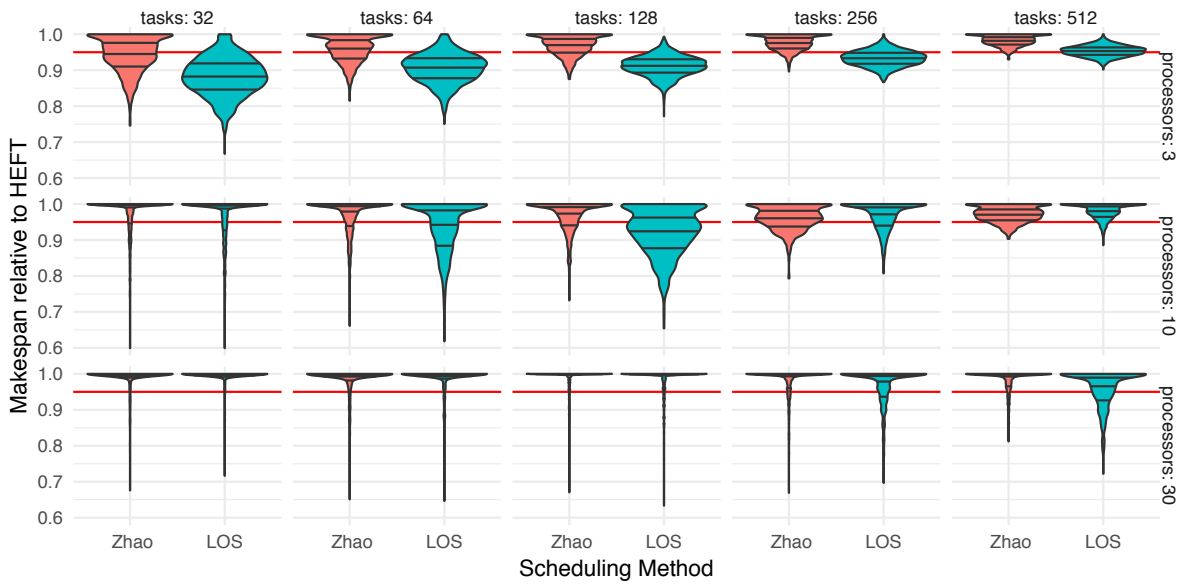

Figure 3.5.: Distribution of makespans relative to HEFT's makespan (lower is better) for different numbers of processors and workflow sizes. The baseline method (red) is consistently outperformed by LOS on three processors. The violin plots summarize both the distribution of the relative makespans and their quartiles, as indicated by the three horizontal lines within each colored area. The red reference line indicates 95% of HEFT's makespan.

Performance Variance

Since LOS uses randomization, it is also interesting to analyze the amount of variation in the relative makespans returned across different runs of LOS. In this section, the variation across three runs for each input instance is analyzed. Mean and standard deviation are computed across the three runs, and the medians of both the means and the standard deviations are reported in Table 3.1.

To exemplify the calculation, consider the following example. In 50% of the experiments with 128 tasks on ten processors, the mean relative makespan seen across three runs was ≤ 0.932 with a standard deviation of ≤ 0.003. Three LOS runs on a specific input instance with 128 tasks and ten processors gave makespans of 310, 306, and 297. The makespan of HEFT's schedule was 347. The relative makespans delivered by LOS are thus 0.89, 0.88, and 0.85, giving a mean relative makespan of 0.877, and the standard deviation across the relative makespans is 0.019. In this case, the mean is below the median mean of 0.932, and the standard deviation is above the median standard deviation of 0.003. Since input instances cannot easily be ordered with respect to both quantities, the medians reported in Table 3.1 are computed independently, one over the average relative makespans and one over the standard deviations of the relative makespans.

Tasks	3 Processors LOS	Zhao	10 Processors LOS	Zhao	30 Processors LOS	Zhao
32	**0.884 ± 0.006**	0.954	1.000 ± 0.000	1.000	1.000 ± 0.000	1.000
64	**0.908 ± 0.008**	0.968	**0.964 ± 0.000**	0.995	1.000 ± 0.000	1.000
128	**0.912 ± 0.006**	0.975	**0.932 ± 0.003**	0.985	1.000 ± 0.000	1.000
256	**0.933 ± 0.005**	0.979	0.983 ± 0.004	**0.964**	**0.996 ± 0.000**	1.000
512	**0.954 ± 0.004**	0.984	0.986 ± 0.005	**0.974**	**0.979 ± 0.000**	1.000

Table 3.1.: Median values of the mean and standard deviation of LOS relative makespans over three runs on each input instance. © 2018 IEEE

3.2.4. Progress and Convergence

This section analyzes the progress made by LOS during its adaptive search for a schedule. The following metrics are reported: average improvement ratio, the number of improvements, and the latest improvement time.

Average Improvement Ratio

The average improvement ratio is the average of the improvement ratios obtained between exploitation phases during a LOS run. The improvement ratio is the ratio of the makespan of the best improvement π_L^* and the makespan of the current best solution at that time. For instance, when improving the makespan in two exploitation phases first from 100 to 90 and then from 90 to 45, the first improvement ratio is 0.9, and the second is 0.5, giving an average improvement ratio of 0.7. The average improvement ratio indicates whether the method finds better schedules using a sequence of small improvements or using fewer but larger improvements.

The median average improvement across all experiments is 0.95, considering only the cases where LOS finds an improvement over HEFT. Figure 3.6 summarizes the distribution of average improvement ratios for all combinations of dag sizes and processors. For small dags on few processors, larger average improvement ratios are found. The distributions shift towards smaller average improvement ratios with increasing dag size. Note that some of the diagrams, e. g., for 30 processors and 32 tasks, summarize fewer input instances, as the experiments for which no improvement has been found have been removed, as explained in greater detail in the next paragraph.

Number of Improvements

The number of improvements is the number of exploitation phases that yield a better solution than the reference solution. It can be zero if none of the exploitation phases finds a schedule that is better than the initial solution. The number of improvements combined with the average improvement ratio gives an estimate of the overall improvement. Figure 3.7 shows the distribution of the number of improvements over all experiments. Input instances in which the best solution was better than HEFT's solution are shown in green. Instances in

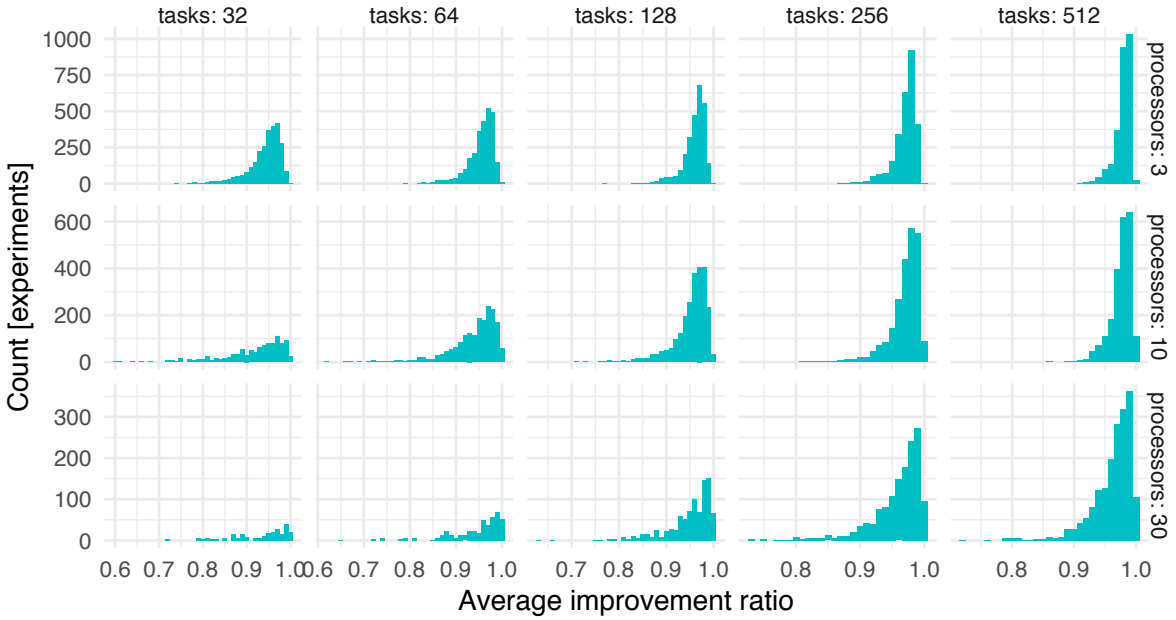

Figure 3.6.: The average improvement ratio between exploitation phases. Small ratios indicate that solutions are improved using small improvements.

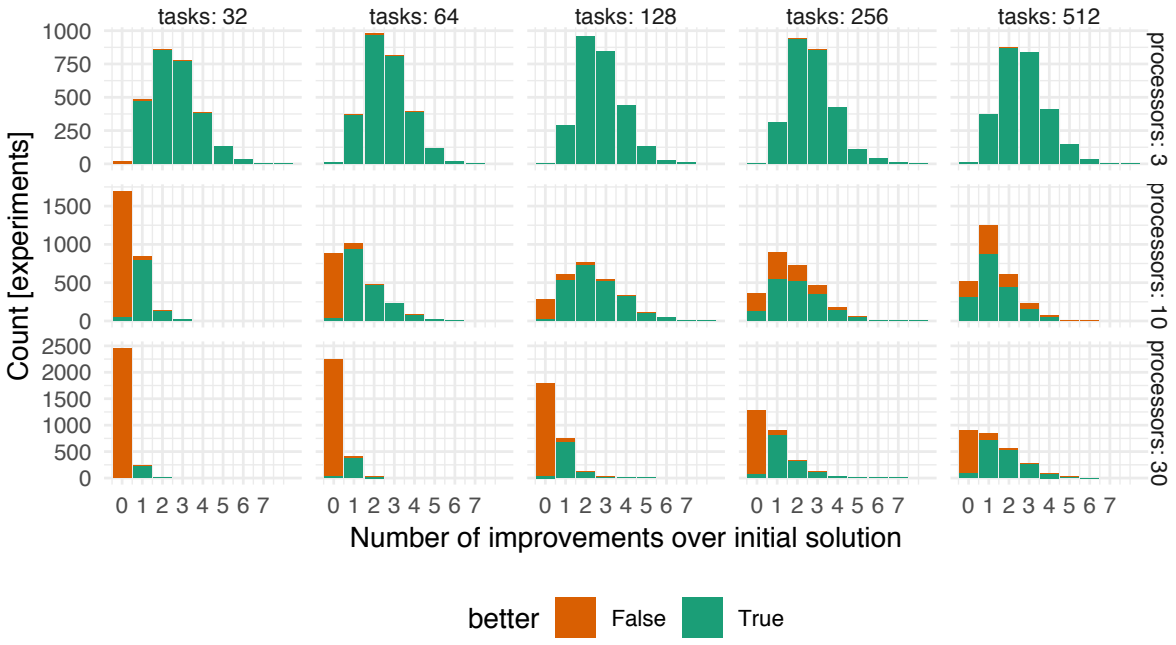

Figure 3.7.: The number of improvements refers to the number of times LOS was able to improve upon its current best solution. The *better* variable refers to whether LOS was able in a given experiment to find a schedule that outperforms HEFT.

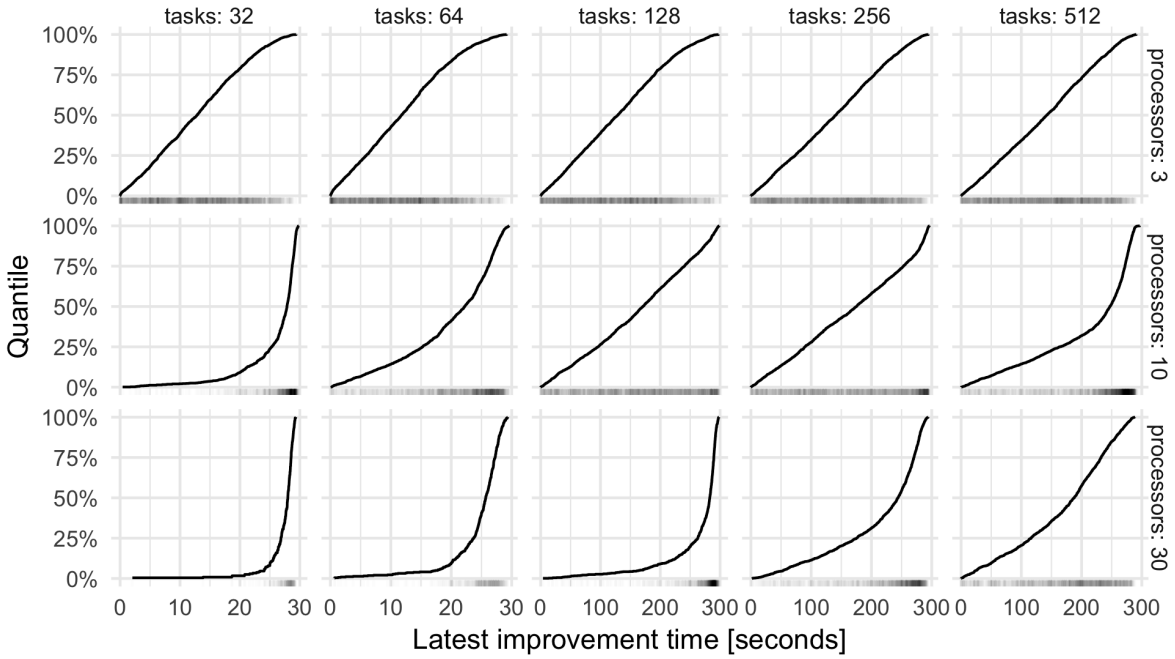

Figure 3.8.: Cumulative distribution functions of the latest improvement time, i. e., the elapsed wall clock time when LOS was last able to improve upon its previous solution. Only experiments for which at least one improvement was found are included.

which LOS did not find a better schedule than HEFT's schedule are shown in red. This was observed in 60 out of 13440 cases for three processors but happened more frequently for larger processor counts. In some cases, e. g., for 32 tasks on 30 processors, the input instances might be too simple, but in general, adding more processors seems to result in significantly harder input instances. Among the unsuccessful runs, some runs with a non-zero number of improvements have been observed, e. g., for 512 tasks on 30 processors. This indicates that LOS makes progress, but not fast enough to outperform HEFT within the given time budget.

Latest Improvement Time

The latest improvement time refers to the elapsed wall clock time until LOS does not find more improvements. This metric indicates at which portion of the allocated time budget LOS converges to its solution. For instance, a latest improvement time of 50 seconds on a time budget of one minute indicates that improvements are found close until the budget expires, which suggests that LOS could further improve on a larger time budget. Figure 3.8 shows the distribution of latest improvement times. For large input instances, LOS has been allotted a time budget of five minutes. For the smaller input instances, a time limit of 30 seconds was set. For three processors, convergence times are relatively uniform. For input instances with ≤ 64 tasks on 30 processors, more than 80% of experiments converge almost immediately, and only a few show improvements towards the end of the budget. In some

configurations, e. g., 128 tasks on 30 processors, LOS seldom converges before the end of the time budget. This indicates that schedules could further be improved by increasing the time budget.

3.2.5. Non-significant Factors

A few alternative designs have been evaluated, which showed very similar behavior to LOS:

- The effect of deriving an initial L-Order from HEFT's default ordering (averaged computation and communication times with upward rank) was tested. Such an ordering can be computed by sorting the tasks in each level according to HEFT's ranks. The idea is to provide a better starting point than random permutations of the levels. However, improvements have been marginal.
- A simpler processor assignment method was evaluated. The simpler method always appends tasks to the end, instead of using HEFT's insertion-based policy that looks for gaps between already scheduled tasks. The idea is to reduce the time needed to compute a schedule for a single input instance, which is a limiting factor on the number of evaluated solutions when facing a time budget. The question was whether LOS could replace the insertion mechanism by considering more orders, orders that may not require insertion in the first place. However, both methods did not yield significant differences, such that HEFT's default (but slower) insertion-based policy was used in the experiments.[2]
- The performance of LOS was evaluated for different rejection probabilities $p \in \{1\%, 5\%, 20\%\}$ for switching between $p_{\mathcal{N}}(m \leq r)$ and $p_r(m \leq r)$ (Equation 3.4). No significant differences were observed, which indicates that the method is robust with respect to this parameter.

3.3. Related Work

Improving upon HEFT's performance is an ongoing effort [Pietri and Sakellariou, 2019, Huang et al., 2014, Shetti et al., 2014, Bittencourt et al., 2010]. Several alternative weighting schemes have been proposed, which are briefly summarized below. In addition, several alternatives to HEFT have been proposed. Since these usually compare to HEFT as the baseline, the evaluation focused on the performance of LOS relative to HEFT.

As described in Section 2.2.1, HEFT uses averaged computation and communication times before computing upward ranks. In [Zhao and Sakellariou, 2004], it has been empirically shown that the weighting method can have a significant impact on the makespan of HEFT's

[2] The impact of using the insertion-based policy on the performance of the original HEFT algorithm was also evaluated. In 20% of the cases, using the insertion-based policy resulted in 1%-10% shorter schedules compared to simply appending tasks at the end of a processor's partial schedule. In about 5% of the cases, insertion-based scheduling even resulted in longer schedules (by up to 10%), and it did not change the makespan in the remaining cases.

schedule. The authors evaluated simple alternative weighting methods, such as the worst-case computation time $\max_p w(T_i, p)$ to compute the weights of tasks and edges. In contrast to their method, LOS searches alternative orders rather than trying to come up with a single weighting method that results in a favorable order most of the times. In addition, it considers variations of orders that affect only certain parts of the dag, which is similar to applying different weighting methods to different parts of the dag.

In [Huang et al., 2014], a weighting method for series-parallel graphs is proposed. Instead of using the maximum of the child weights, the child weights are summed up. The idea is to consider more than the critical path to assess the downstream work of a task. However, the authors focus on a particular class of dags, whereas the proposed method does not impose restrictions.

In [Ilavarasan et al., 2005], a weighting method is proposed that adds the sum $\sum_{(u,v)\in E} c(u, v)$ of outgoing data transfers from a task u to its weight. It is assumed that the network links have the same speed, although averaging could be used to handle heterogeneous transfer speeds. The authors do not report explicit numbers, but the Figures indicate that the schedules are about 3%-7% faster than HEFT's.

In [Shetti et al., 2014], a weighting method using a sufferage metric [Casanova et al., 2000] is proposed. Sufferage refers to the increase in computing time when scheduling a task on the second-best processor compared to the fastest processor for that task. Using sufferage allows for prioritizing tasks that would potentially run much slower when scheduled with lower priority. The scheme is used to apply HEFT in CPU-GPU environments.

In [Sakellariou and Zhao, 2004], the BMCT heuristic is proposed, which is supposed to be more robust than HEFT concerning the choice of the weighting method. First, HEFT's weighting method (averaging) is applied. Then, the dag is partitioned into a sequence of independent task sets, which are then scheduled using a heuristic for independent task scheduling. The method is computationally more expensive than HEFT because it iteratively optimizes the placement of the tasks in each set. The study reports that HEFT's makespan was, on average, 3.25% longer than the best solution produced by an ensemble of BMCT and five other methods. Depending on the configuration, LOS produces up to 11.6% shorter makespans than HEFT on median. However, this also depends on the graphs used for evaluation. Sakellariou et al. used, among others, the method described in [Zhao and Sakellariou, 2004] to generate random graphs.

In [Bittencourt et al., 2010], a lookahead version of the HEFT algorithm is proposed that considers the earliest finish times of the children of a task for a given placement of the task, in addition to the task's own earliest finish time. In [Arabnejad and Barbosa, 2014], an improved version of the lookahead-HEFT is proposed that achieves similar effects at lower computational costs.

3.4. Discussion

LOS employs an innovative randomized approach to dag scheduling on heterogeneous resources with communication times. It uses L-Orders to structure the space of possible orderings into regions and estimates the probability for various regions of randomly sampling

a better order than the current best order. LOS schedules for three processors and less than 256 tasks outperform HEFT by at least 10% in 50% of the experiments, whereas the baseline method gives only a median 3% improvement. However, in some scenarios both LOS and the baseline method find improvements over HEFT only rarely, e. g., for workflows with 128 tasks on 30 processors. Although there is no clear relationship between task-to-processor ratio and the performance of LOS, improving over HEFT seems to become harder on infrastructures with more processors.

A limitation of the method is that it is tailored to highly data-parallel workflows. In the extreme case of a sequential workflow that has only one task per level, there is only one topological order, which implies that LOS has no room to vary task priorities and will always return the same schedule as HEFT. However, better approaches exist for this case: An optimal schedule can be found using dynamic programming [Kougka et al., 2017]. The following gives an overview of research options and improvement opportunities for LOS.

Guided Search

The idea of basing a search-based scheduling algorithm on the principle of list scheduling seems promising. It simplifies the search by restricting possible schedule alterations to changing the priority of a task. However, the shuffle operation is a relatively coarse-grained way to modify task priorities, and the following improvements seem promising:

- Fine-grained modifications. Partitioning the graph into levels restricts the extent of the shuffle operation to a certain degree but suggests further optimizations. One could start with the proposed partitioning into levels and later refine or change the partitioning. Refinement preserves the advantage of the level-based approach, which guarantees task priorities that result in a topological order. An alternative to refinement would be fine-grained operators, such as swapping the priority of two tasks. Adding operators leads to a more evolutionary optimization approach, which has the disadvantage of necessitating additional hyperparameters that govern the selection of operators.

- Intelligent modifications. The shuffle operator is oblivious to task properties such as the amount of downstream work, which drive the prioritization in heuristic approaches. Modifying task priorities more intelligently, e. g., by combining a variety of heuristics, would be interesting. An interesting question is whether a mapping from tasks to priorities can learned. Consider using the best mapping from an ensemble of heuristics to initialize a neural network. The network would ideally learn when to apply which heuristic or find entirely new heuristics.

Scalability

The experiments also show scalability limitations of static task graph scheduling. Most related work [Sakellariou and Zhao, 2004, Bittencourt et al., 2010, Arabnejad and Barbosa, 2014] evaluates dag scheduling heuristics on graphs of at most 500 tasks and up to 32 processors only. Even when modeling entire compute nodes as processors instead of CPU cores as processors, this departs by orders of magnitude from large-scale compute clusters

and scientific workflows with thousands of tasks. The problem here is mainly scheduling time, as most scheduling heuristics have a complexity of at least $\mathcal{O}(v^2p)$, where v is the number of tasks and p is the number of processors [Arabnejad and Barbosa, 2014]. In large-scale workflows, partitioning a workflow into sub-workflows to reduce scheduling complexity seems appropriate. However, determining the boundaries of these partitions may not be trivial.

Developing more elaborate data structures and algorithms to re-use partial solutions would improve the performance of the implementation. At the moment, after each modification of priorities, processors are assigned from scratch. Reusing the parts of a schedule that are not affected by priority changes could speed up the construction and evaluation of schedules.

Learning at Run Time

A central problem for practical application is to obtain the information required for planning, such as the execution time of every task on every processor, the amount of output data it produces, and reliable network bandwidth estimates. This information can be very challenging to predict in practical scenarios and is likely to change during workflow execution. Several prior works investigated the impact of uncertain information on the makespan of scheduling heuristics [Hsu et al., 2011, Bittencourt et al., 2012, Zheng and Sakellariou, 2013, Kougka et al., 2015, Malawski et al., 2015, Agullo et al., 2016, Tumanov et al., 2016, Ilyushkin and Epema, 2018]. However, none of these papers addresses the issue of how to obtain estimates for task run times and related planning information. The next chapter proposes to obtain resource usage estimates during workflow execution, which necessitates a different approach to scheduling.

4. Feedback-Based Scheduling of Scientific Workflows

This chapter introduces a feedback-based approach to workflow scheduling. The idea is to learn task resource requirements at run time rather than having a user provide them prior to workflow execution. In the proposed approach, the workflow management system measures task resource usage to update a prediction model during the execution of the workflow, as shown in Figure 4.1. This creates a feedback loop between the scheduling heuristic and the prediction model because the scheduling decisions determine the arrival of new training data, and the updated model's predictions, in turn, affect scheduling decisions. The resulting scenario is new, as resource usage estimates change during workflow execution due to updates of the prediction models.

This chapter assumes the batch execution model of workflow execution, as introduced in Section 2.3. The batch execution model implies a series of significant differences compared to Chapter 3. In the batch execution model, the workflow management system assigns a fixed amount of resources to a task before executing it, introducing the possibility of failures in case of insufficient resources. The focus of the chapter lies on predicting peak memory usage of tasks. In contrast, the classic static task graph scheduling scenario considers only processors and assumes perfect knowledge of task run times.

This chapter uses simulation experiments to study the performance and critical parameters of a workflow management system incorporating online learned resource estimates. System parameters comprise, e. g., the model used for predicting resource usage and the scheduling heuristic. Performance is evaluated with respect to memory allocation quality and workflow execution time. Simulations are conducted using a factorial design, simulating system performance under all combinations of configurations over a corpus of workflows. The experimental design and result analysis focuses on the following issues:

- Comparison of online learned memory usage estimates and user estimates
- Trade-offs between resource efficiency and execution time
- Robustness of configurations with respect to long-tail execution times of workflows
- The potential of tuning the system configuration for a specific workflow

This chapter is structured as follows. Section 4.1 describes the methods used for scheduling and online prediction of task memory usage. Section 4.2 describes the experimental design, including the synthetic workflows used for evaluation. Section 4.3 provides an analysis of the simulation results. Section 4.4 reviews related work and Section 4.5 discusses and contextualizes the results within the broader framework of this thesis.

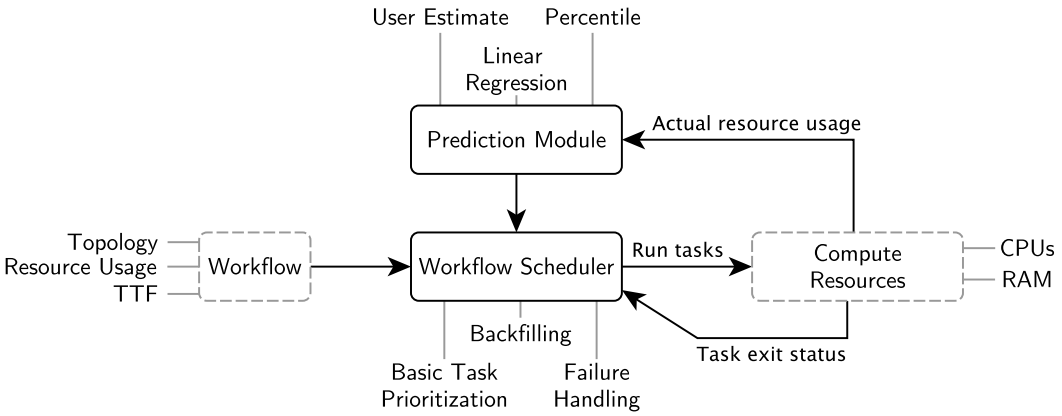

Figure 4.1.: The prediction-scheduling-execution feedback loop. Annotations indicate parameters for the components, such as different task prioritization methods. Dashed lines indicate non-tunable components, compared to the components of the workflow management system for which the parameters can be chosen.

4.1. Method

The methods section describes the scheduling methods (Section 4.1.1) and the prediction methods (Section 4.1.2) used in the simulation experiments.

Figure 4.1 shows the interaction between scheduling and prediction. The simulation receives a workflow as an input, represented as a directed acyclic graph. The simulated workflow management system maintains one prediction model per abstract task. The scheduler consults the corresponding prediction model to obtain a peak memory usage estimate based on the task's input file sizes. Task execution then either succeeds if the estimate is equal to or above the actual peak memory usage; otherwise, the task fails. When a task succeeds, the online learning model is updated with the task's actual peak memory usage. Until the first three tasks succeed, user estimates (see Section 4.2.1) are used.

4.1.1. Scheduling

The workflow management system dynamically schedules tasks. This means that scheduling decisions are made during the execution of the workflow, as opposed to constructing an execution plan prior to the execution of the workflow. When a task slot becomes free on a machine, the workflow scheduler is queried for a task to run on this machine.

In this setting, similar to Chapter 3, the central element of scheduling is assigning priorities to tasks that determine the order in which they are executed. The scheduling heuristics evaluated in this chapter combine different basic task prioritization heuristics (see next section) with different strategies to handle the case of task failures (see next section but one) to determine task priorities. In addition, a backfilling mechanism (see next section but two) allows the scheduler to skip up to $k - 1$ of top-ranking ready tasks if there are insufficient resources to run them. The approach is outlined in Algorithm 3.

```
Callback ScheduleTaskTo(Machine M):
    // Number of tasks to consider for execution
    candidates ← k;
    while ¬ taskQueue.isEmpty() ∧ candidates > 0 do
        // Retrieve highest-ranking ready task
        T ← taskQueue.poll();
        p ← predicted peak memory usage of T;
        if available memory on M ≥ p then
            Start task T on M;
            break;
        end
        else
            unsuitable ← unsuitable ∪ T;
        end
        candidates ← candidates − 1;
    end
    re-insert unsuitable tasks into taskQueue;
end
```

Algorithm 3: To decide which task to run next, a priority queue of ready tasks is polled. In addition, a backfilling mechanism allows to skip the top tasks in the queue if the current machine cannot accommodate the predicted memory needs.

Basic Task Prioritization

Nine task prioritization schemes have been evaluated. The HLF heuristic is inspired by the static scheduling approach in Chapter 3. The LFF, SMF, SML, SIF, and SIL heuristics prioritize tasks according to available training data, predicted memory usage, or input size. As baseline scheduling heuristics, the FIFO and LIFO scheduling heuristics, which are based on task ready times, and a random scheduler have been evaluated. An overview is given in Table 4.1.

- Highest level first (HLF): prioritizes tasks with a high level (Definition 2.8), i. e., tasks with long paths of tasks ahead. This is similar to the upward rank used by the HEFT algorithm (Section 2.2.1), except that unit run times are assumed for all tasks to avoid the need for task run time estimates.

- Least finished first (LFF): prioritizes tasks by the amount of training data available to the prediction model so far, i. e., the number of tasks that have successfully finished so far. The priority of a task is set to the current size n of the resource usage measurement set D that is used to train the peak memory usage prediction model for the given task.

- Smallest predicted memory first/last (SMF/SML): prioritizes tasks according to their predicted peak memory usage $\hat{r}$. SML is similar to the first fit decreasing bin packing strategy, which packs items into bins beginning with the ones taking up the most space. The central idea is, in both cases, to use small tasks to fill gaps left by big tasks.

- Smallest input first/last (SIF/SIL): prioritize tasks by the amount of input data they process, i. e., the sum of their input file sizes. The purpose of this scheduling heuristic is to evaluate possible effects of the order in which training data arrives.
- The first-in-first-out (FIFO) heuristic executes tasks in the order the scheduler detects that they are ready for execution, prioritizing tasks with early ready times.
- The last-in-first-out (LIFO) heuristic prioritizes the tasks that have most recently become ready for scheduling. This is similar to a depth-first search through a graph.
- The random scheduling heuristic (RND) assigns random priorities to tasks.

Failure Handling Strategies

In addition to the basic task prioritization schemes from the previous section, it is important for a scheduler to handle the case of task failures due to insufficient requested memory. The number of times a task has previously failed provides some information on the performance of the prediction model. Intuitively, a prediction model that underpredicts memory usage more often than expected either lacks training data or is not able to fit the data well.

- Persevere strategy. In the case of insufficient training data, it might be advantageous to prioritize the tasks to increase the rate at which training data is produced. If the prediction model fits the data poorly, it could also be advantageous to prioritize tasks that have previously failed, because the poor prediction quality will force the tasks to stay longer in the system before completion, as tasks have to be retried with an increased allocation. Prioritizing tasks that have previously failed is referred to as the *persevere* failure handling strategy.
- Postpone strategy. On the other hand, decreasing the priority of tasks that have failed before offers a chance that new training data arrives in the meantime, which could increase the quality of the predictions. Lowering the priority of previously failed tasks is referred to as the *postpone* failure handling strategy.

The scheduler sorts ready tasks first by the number of previously failed attempts and then, within each group, by the basic task prioritization criterion. Depending on whether tasks with previous failures are prioritized or delayed, the failure handling strategy is referred to as persevere or postpone, respectively. As a baseline, the ignore failure handling strategy sorts tasks only by their basic task priority criterion.

Backfilling

The backfilling mechanism (see Algorithm 3) allows the scheduler to skip up to $k-1$ of the top-ranking ready tasks if there are insufficient resources to run them. Here, k is a parameter that balances adherence to task priorities and resource utilization. In the experiments, $k \in \{1, 5, 15\}$ was chosen, where $k = 1$ corresponds to no backfilling.

Basic Task Prioritization	Acronym	Priority criterion
Highest level first	HLF	level(T_i)
Least finished first	LFF	$-n$
Smallest input first	SIF	$-x_i$
Smallest input last	SIL	x_i
Smallest predicted memory first	SMF	$-\hat{r}_i$
Smallest predicted memory last	SML	$\hat{r}_i$
First-in-first-out	FIFO	$-$ready time(T_i)
Last-in-first-out	LIFO	ready time(T_i)
Random	RND	$\mathcal{U}(0,1)$

Table 4.1.: The nine basic task prioritization criteria evaluated in the simulation experiments. Tasks with high priority values are started before tasks with low priority values.

For instance, in case of the highest level first heuristic, it might be better to leave resources idle to be able to start a task on the critical path of a workflow sooner. For the random scheduling heuristics, on the other hand, strictly enforcing task priorities is expected to perform worse than using backfilling to improve utilization.

4.1.2. Online Peak Memory Usage Prediction

The idea of feedback-based workflow scheduling requires online prediction models that use peak memory usage measurements from the tasks that have finished so far to predict peak memory usage for the remaining tasks in the workflow. One prediction model is trained per abstract task. Every time a task completes successfully, the model of the corresponding abstract task is updated with the observed input size and peak memory usage. The updated model is then consulted by the scheduler to obtain a prediction of the peak memory usage of ready tasks. When a task fails due to insufficient allocated resources, the allocated memory is repeatedly doubled until the task succeeds (exponential re-allocation, see Definition 2.22).

The experiments evaluate two types of prediction models: (1) percentile estimators and (2) conservative variants of linear regression. These prediction models are simple to implement in any environment, as well as being fast to update online.

When predicting peak memory usage of tasks, the distribution of the prediction errors is of central importance, because underestimating the peak memory usage of a task results in task termination and overestimation does not. Thus, a prediction model's tendency to underpredict peak memory usage should be controllable.

A simple measure to quantify the tendency of a prediction model to underpredict is the failure rate, i. e., the fraction of predictions that are smaller than the actual peak memory usage. More complex characterizations of the error distribution of a peak memory usage prediction model are discussed in Chapter 5. For instance, the magnitude of the prediction errors affects the amount of wasted resources, which is not captured by the failure rate.

Percentile Predictor

The percentile predictor (PC) predicts the peak memory usage of a task to be a specified percentile of the peak memory usages of all successful tasks so far. Let D be a resource usage measurement set (Definition 2.21, page 24) with n tasks. The predicted peak memory usage $\hat{r}$ for an unseen task is the q-the percentile of the peak memory usages r_i in D.

If all tasks are sampled independently from the same population, the failure rate of the percentile predictor will converge to the chosen percentile as n goes to infinity. However, as the sample percentile is an estimator of the population percentile, the failure rate is not guaranteed to match the chosen percentile, especially for small n. In the simulations, different percentile predictors ranging from the median (PC 50) up to the maximum (PC 100) have been evaluated.

Conservative Linear Regression

In practice, predictions of peak memory usage should take into account additional information about the task. Here, peak memory usage is predicted based on the amount of input data x, i. e., the sum of a task's input file sizes. Ordinary least squares (OLS) regression is a straightforward approach to solve this problem. To obtain control on the failure rate of the regression, a dynamically computed offset d is added to the ordinary least squares prediction, resulting in a linear regression prediction model with a controllable degree of conservatism. The model is updated as follows:

1. Update an OLS regression with the new observation of input size and peak memory usage
2. Compute the prediction errors $r - \hat{r}$ of the new OLS model for all observations
3. $d \leftarrow$ q-th percentile of the prediction errors
4. Add d to the intercept of the model

By adding the q-th percentile of the prediction errors, the predictions of the model are shifted, such that afterwards $\hat{r} \geq r$ for q percent of the training data. The predicted peak memory usage for an unseen task equals

$$\hat{r} = \theta_1 x + \theta_0 + d \tag{4.1}$$

where θ_1 is the slope and θ_0 is the intercept of the OLS regression model. Updating an ordinary least squares regression model with a single input and a single output can be done incrementally in $\mathcal{O}(1)$. Computing d is in $\mathcal{O}(n \log n)$ for n observations because updating predictions and prediction errors for the available training data is in $\mathcal{O}(n)$ and computing the percentile of the errors is in $\mathcal{O}(n \log n)$. While updating the slope and intercept of the regression model is cheap, computing d every time a new observation is added raises the computational complexity of adding n samples to $\mathcal{O}(n^2 \log n)$. To reduce the cost, d is computed only when n has increased by 5% since the last computation of d, resulting in a cost of $\mathcal{O}(n \log^2 n)$.

4.2. Experimental Design

This section covers the design of the simulation experiments used to evaluate the feedback-based scheduling approach and its scheduling and prediction parameters. Discrete event simulation has been used to simulate the execution of the workflows. The DynamicCloudSim simulation framework [Bux and Leser, 2015] has been extended, which itself is an extension of the CloudSim framework [Calheiros et al., 2010]. DynamicCloudSim adds workflow execution functionality to CloudSim, such as data structures for workflows and algorithms to schedule them. Simulation aspects such as main memory as a compute resource, custom schedulers, and online prediction models have been added to the framework. Workflow execution is simulated on a shared resource pool of 128 CPU cores and 1952 GB of main memory.

Performance is analyzed with respect to two criteria, execution time and memory allocation quality (MAQ, see Section 2.3.3). Since the best possible execution time depends on the workflow, makespans are normalized relative to a lower bound (see Section 4.2.1). This is referred to as makespan ratio (MSR).

The following subsections describe the method used to generate a corpus of workflows for evaluation and summarizes the simulation parameters. Table 4.2 summarizes the values used in the experiments.

4.2.1. Synthetic Workflows

A workflow generator based on the Pegasus Workflow Generator [Ferreira da Silva et al., 2014] was used to generate an evaluation workload comprising five types of workflows: Genome, Cybershake, Sipht, Montage, and Ligo. The workflow types differ in the number of abstract tasks, the number of tasks per abstract task, and their topology, i. e., the dependency patterns between tasks. Background information on the applications behind these workflows and exemplary topologies can be found in [Juve et al., 2013]. The Pegasus workflow generator was extended with a sampling method to generate correlated input sizes and peak memory usage values.

The sampling method for the input sizes and peak memory values is based on two assumptions.

1. Memory usage is assumed to be heterogeneous, both within and across abstract tasks [Juve et al., 2013, Singh et al., 2017].

2. It is assumed that parts of the heterogeneity can be explained by input sizes, frequently in the form of a linear model [Bux et al., 2017].

Peak memory usages are sampled from joint normal distributions of input sizes and peak memory values to implement these assumptions, subsequently referred to as random memory model. Each abstract task is associated with a different random memory model, the parameters of which are chosen randomly and independently. With a chance of 50%, an abstract task exposes a linear relationship to the sum of input file sizes (linear random memory model, see next section). Otherwise, input size and peak memory consumption are sampled independently. While the independent case is trivial, sampling from a joint probability distribution that exposes linear relationships between the variables is described in the following.

Linear Random Memory Models

This section outlines the process of creating a random resource usage measurement set (Definition 2.21). The input sizes x_i and peak memory usages r_i are modeled as linearly related, normally distributed random variables X and Y, respectively, according to Equation 4.2.

$$Y \sim \theta_1 X + \theta_0 + \mathcal{N}(0, \sigma_e^2) \tag{4.2}$$

The error term $\mathcal{N}(0, \sigma_e^2)$ adds normally distributed variation in memory consumption that is not explained by input size. For better control of the peak memory usages, the sampling starts from a desired mean μ_y and standard deviation σ_y for the peak memory consumption values. Let the input sizes and peak memory usages be normally distributed. Then, Equation 4.2 can be used to derive the mean μ_x and standard deviation σ_x of the input distribution that gives the desired μ_y and σ_y. To sample from the joint distribution, input sizes are sampled from a normal distribution $\mathcal{N}(\mu_x, \sigma_x^2)$ and then transformed according to Equation 4.2. Let $X \sim \mathcal{N}(\mu_x, \sigma_x^2)$ denote the random variable describing the distribution of input sizes. Since the sum of independent normally distributed random variables is also normally distributed

$$\begin{aligned} Y &\sim \theta_1 \mathcal{N}(\mu_x, \sigma_x^2) + \theta_0 + \mathcal{N}(0, \sigma_e^2) \\ &= \mathcal{N}(\theta_1 \mu_x, \theta_1^2 \sigma_x^2) + \theta_0 + \mathcal{N}(0, \sigma_e^2) \\ &= \mathcal{N}(\theta_1 \mu_x + \theta_0, \theta_1^2 \sigma_x^2) + \mathcal{N}(0, \sigma_e^2) \\ &= \mathcal{N}(\underbrace{\theta_1 \mu_x + \theta_0}_{=\mu_y}, \underbrace{\theta_1^2 \sigma_x^2 + \sigma_e^2}_{=\sigma_y^2}) \end{aligned}$$

Solving μ_y for μ_x and σ_y^2 for σ_x^2, one obtains

$$\mu_x = \frac{\mu_y - \theta_0}{\theta_1} \tag{4.3}$$

$$\sigma_x^2 = \frac{\sigma_y^2 - \sigma_e^2}{\theta_1^2} \tag{4.4}$$

Rather than choosing σ_e^2 directly, it is more convenient to choose the amount of peak memory usage variance explained by input size, i. e., the coefficient of determination R^2. To do so, the desired variance σ_y^2 is split into explained variance $R^2\sigma_y^2$ and unexplained variance $(1-R^2)\sigma_y^2$. The error variance is set to the amount of unexplained variance:

$$\sigma_e^2 = (1 - R^2)\sigma_y^2 \tag{4.5}$$

Substituting σ_e^2 into 4.4 gives

$$\sigma_x^2 = \frac{R^2 \sigma_y^2}{\theta_1^2} \tag{4.6}$$

Selecting the input variance σ_x^2 according to the explained variance $R^2\sigma_y^2$ and the error variance according to unexplained variance $(1 - R^2)\sigma_y^2$ results in the chosen Pearson correlation coefficient $\rho_{X,Y} = R$ (see Appendix C for a proof). This allows for varying the predictability

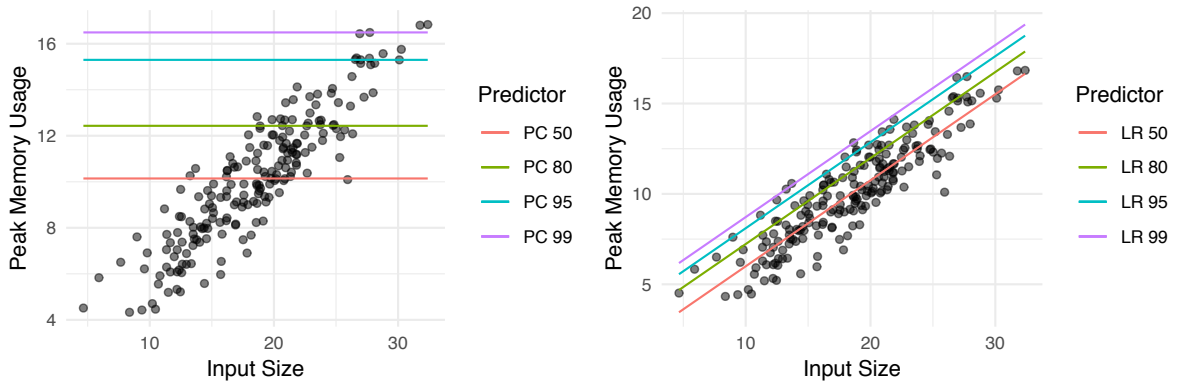

Figure 4.2.: Example data generated using the linear random memory model from Section 4.2.1 with parameters $\mu_y = 10, \sigma_y = 3, R^2 = 0.8, \theta_1 = 0.5, \theta_0 = 1$. The colored lines show predicted peak memory usage as a function of input size for the percentile prediction model (left) and the conservative linear regression model (right).

of memory usage from perfect, for $R^2 = 1$, to completely unrelated, for $R^2 = 0$. In the experiments, the parameters of the linear random memory models have been sampled according to Table 4.2. Figure 4.2 shows an example of data generated with a linear random memory model.

The run times τ_i of the tasks are generated by the Pegasus Workflow Generator to resemble measurements from real-world workflows. The run times in case of insufficient resource allocation τ_i^* are set to a fraction TTF $\in \{0.1, 0.5, 0.9\}$ of the run time τ_i. Each workflow consists of approximately[1] 1000 tasks. After generating input sizes and peak memory usage values for each task, the task run times are scaled such that the sum of the products of task run times and memory consumption in each workflow equals one terabyte-week. Scaling the overall resource is convenient for the evaluation because it makes workflows more comparable, as explained in the next section.

Lower Bounds on Makespan

Lower bounds help to assess the execution time of a workflow, i. e., to understand whether it signifies good or poor performance of the system. Due to the NP-completeness of the scheduling problem, computing the minimum execution time for a given workflow and execution environment is impractical. However, lower bounds on makespan have been computed for the synthetic workflows by combining three approaches: (1) throughput limits (2) critical paths, and (3) workflow partitions.

Total Work

A first lower bound on workflow execution time can be derived by considering the maximum throughput provided by the available compute resources. Let MEM denote the available

[1]Due to topology constraints, Sipht workflows have 968 tasks, and Genome workflows have 997 tasks.

amount of main memory. A lower bound on the total execution time of a workflow can be derived by dividing the sum of the products of all task run times and their peak memory usage by the amount of available main memory. Let CPUs denote the number of available CPU cores. Another lower bound on the workflow execution time can be derived by dividing the sum of all task run times by the number of available CPU cores. The resource that provides the higher lower bound is the bottleneck resource for the workflow and the according lower bound is referred to as the total work (TW) lower bound on workflow execution time.

$$\mathsf{TW} = \max\left(\frac{1}{\mathsf{MEM}}\sum_i r_i\tau_i, \frac{1}{\mathsf{CPUs}}\sum_i \tau_i\right) \tag{4.7}$$

Critical Path

The TW lower bound assumes perfect parallelization, which is usually not achievable due to dependencies between tasks. A lower bound that takes into account workflow topology can be derived by summing up the run times of the tasks on a critical path through the workflow. Let LP be a path that maximizes the sum of run times of the tasks on the path. The makespan of the workflow cannot be lower than the length of the path since all tasks on the path have to be executed in sequence. This is referred to as the critical path (CP) lower bound on workflow execution time.

$$\mathsf{CP} = \sum_{T_i \in \mathsf{LP}} \tau_i \tag{4.8}$$

Workflow Partitioning

The total work and critical path lower bounds can be combined to construct a lower bound that is at least as tight as the individual lower bounds by partitioning the workflow in a sequence of sub-workflows. Independently of the way a lower bound is derived, it is generally the case that when executing two workflows in sequence, a lower bound on the makespan of the two workflows can be derived by summing up the lower bounds of the individual workflows. If the tasks of a workflow can be partitioned into a sequence $S_1, ..., S_n \subseteq V$ of subsets of the tasks such that every task in set S_i can be started only after every task in S_{i-1} has finished, then the workflow can be treated as a sequence of workflows. In this case, the total work and critical path lower bounds can be computed for the subgraphs induced by the task sets S_i and summed up to obtain a lower bound for the overall workflow, which is here referred to as partition lower bound (PB). An algorithm to partition a workflow and an example is given in Appendix D.

$$\mathsf{PB} = \sum_{i=1}^{n} \max(\mathsf{TW}(S_i), \mathsf{CP}(S_i)) \tag{4.9}$$

Hypothetical User Estimates

An interesting question is how the performance of a system relying on online learned peak memory usage compares to the performance of a system relying on users to estimate peak memory usage. Two methods are used to generate user estimate-like predictions for the corpus of synthetic workflows used in the evaluation:

- Q99. Return the 99th percentile of the true peak memory usages across an abstract task. This approach reflects the observation that users tend to provide conservative estimates in practice [Delimitrou and Kozyrakis, 2014]. It also assumes that users have a good knowledge about the peak memory usages of the tasks since the method uses the actual peak memory usages, a ground truth that is not easy to obtain. This is typically the case in a scenario where users invest time to benchmark and analyze the resource usage of programs, as reported in the case study in Chapter 5. Using the 99th percentile instead of the maximum corresponds to the case where out of memory failures cannot be completely avoided. Preliminary simulation results and related work [Tovar et al., 2018] also show that the maximum tends to be overly conservative and typically delivers worse performance than an estimator of the 99th percentile.

- Power 2. Round the true maximum memory usage for an abstract task to the nearest power of two, in Gigabytes. For instance, if the actual maximum peak memory usage across the tasks of an abstract task is 4.1 GB, the estimated peak memory usage for the tasks belonging to the abstract task is assumed to be 4 GB. This reflects the observation that user estimates are typically coarse-grained and round numbers [Reiss et al., 2012][2].

4.2.2. Simulation Parameters

This section introduces basic terms used in the analysis of the results, namely configuration, workflow, and simulation.

Configuration

A configuration is a combination of a prediction model and a scheduling heuristic, as specified by basic task priority, failure handling, and backfilling parameter. These are the system parameters that can be controlled, whereas the workflows to be executed are assumed to be given. In practice, the amount of resources is also a potential configuration option. Here, the amount of resources is assumed to be fixed, corresponding, for instance, to a scenario where all available resources are used for the execution of a single workflow.

- A total of 14 prediction models have been evaluated: six parameterizations each of the LR and PC models and two constant prediction models, Power 2 and Q99.

[2] "[...] the amount of CPU or memory requested for each task, are very discrete and unsmooth, apparently due to human factors."

- A total of 81 scheduling parameterizations was evaluated, combining each of the nine basic task priorities with the three failure handling strategies and three values for the backfill parameter.

Combining all prediction models with all scheduling parameterizations gives 14·81=1 134 configurations. In addition, a perfect prediction model that exactly predicts the peak memory usage of a task was evaluated in combination with the 81 scheduling parameterizations. The resulting 81 configurations are used as theoretical reference configurations simulating perfect knowledge of task memory usage. In the following, the 1 134 realistic configurations will briefly be referred to as *configurations*, while the 81 configurations involving known peak memory usages will be referred to as *reference configurations*.

Workflow

A workflow is a combination of a workflow type, a random seed, and a value specifying the time to failure for all tasks.

- Workflows of five types have been generated, Cybershake, Genome, Ligo, Montage, and Sipht [Juve et al., 2013]. The workflow type specifies the abstract tasks and workflow topology.
- 100 random workflow instances have been generated for each workflow type. The workflow instance specifies the run times and peak memory usages of the tasks.
- The run time in case of insufficient resources is specified by the time to failure parameter, which has been varied across three levels, 0.1, 0.5, and 0.9.

This results in a total of 5·100·3=1500 workflows. Simulating the execution of each workflow under each configuration results in 1 822 500 simulated workflow executions, or simulations for short. 1 701 000 simulations use realistic configurations, and 121 500 simulations use the reference configurations.

4.3. Experimental Results

This section reports and discusses the results from 1 822 500 simulated workflow executions, amounting to roughly 4 400 CPU hours. The analysis focuses on two main cases: (1) Executing all workflows using the same configuration and (2) selecting the best configuration for each workflow.

4.3.1. Fixed Configurations

In this section, the performance of different configurations averaged over all workflows is evaluated. This corresponds to the scenario where all workflows are executed using the same configuration, i. e., prediction model and scheduling parameters. In Section 4.3.2, the potential of selecting the best configuration for individual workflows is evaluated. In addition

Table 4.2.: Simulation Parameters

	Parameter	Value
Workflows	Type	Cybershake, Genome, Ligo, Montage, Sipht
	Instances	100 (per type)
	Tasks	1000 per workflow
	Time to failure	0.1, 0.5, 0.9
Memory Models	Average	$\mu \sim \mathcal{U}(1\text{GB}, 500\text{GB})$
	Coefficient of Variation	$c_v \sim \mathcal{U}(0.3, 0.6)$
	Coefficient of Determination	$R^2 \sim \mathcal{U}(0.25, 0.75)$
	Slope	$\theta_1 \sim \mathcal{U}(0.2, 2)$
	Intercept	$\theta_0 = 0$
Prediction Models	Percentile (PC)	percentile $\in \{50, 80, 95, 97, 99, 100\}$
	Conservative Linear Regression (LR)	percentile $\in \{50, 80, 95, 97, 99, 100\}$
	User Estimate	Power 2, Q99
Scheduling	Task Priority	HLF, LFF, SIF, SIL, SMF, SML, FIFO, LIFO, RND
	Failure Handling	Persevere, Postpone, Ignore
	Backfilling	$k \in \{1, 5, 15\}$
Resources	Cores	128
	RAM	1952 GB

Predictor	Priority	Failures	k	**Mean MSR**	99% MSR	Mean MAQ	1% MAQ
LR 80	HLF	Persevere	5	2.160	3.384	57.0%	40.5%
LR 80	FIFO	Persevere	15	2.160	3.489	56.7%	40.1%
LR 80	SML	Persevere	15	2.167	3.536	56.7%	41.2%

Table 4.3.: Top 3 configurations with respect to average makespan ratio. 99% MSR denotes the 99th percentile of makespan ratio across all workflows. 1% MAQ denotes the long tail performance with respect to memory allocation quality.

to average MSR and MAQ, the long tail performance is considered by computing the 99th percentile of the makespan ratio across all workflows. First, an overview of the average performance across different configurations is given. Second, top-performing configurations are identified, and trade-offs between speed and resource utilization are discussed.

Performance Range

The average makespan ratio ranges from 2.16 for the configuration that is fastest on average to 3.89 for the configuration that is slowest on average. Average memory allocation quality ranges from 44.5% for the least resource-efficient configuration to 57.4% for the most resource-efficient configuration. This means that on average, only around 50% of the allocated resources are used, which results in increased makespan ratios. For comparison, for the 81 reference configurations, i. e., configurations achieving 100% memory allocation quality, average makespan ratios range from 1.19 to 1.3, which is much closer to the theoretical lower bound.

The performance of the 1134 configurations in terms of average and 99th percentile makespan ratio and average memory allocation quality is shown in Figure 4.3. The performance criteria are highly correlated: configurations that have lower memory allocation quality tend to have higher makespan ratios. The two quantities are negatively, strongly, and significantly correlated (Pearson $r = -.902, p < 0.001$). Average makespan ratio and 99th percentile makespan ratio are also strongly and significantly correlated ($r = 0.99, p < 0.001$).

Top Performing Configurations and Pareto Optimal Configurations

Table 4.3 shows the three configurations that deliver the best average makespan ratio. The best performing basic task priorities are FIFO, HLF, and SML, all using the persevere failure handling strategy and larger backfilling parameters. Their average performance with respect to both performance criteria is very similar but differs slightly in robustness. For HLF, 99% of the workflows have a makespan ratio of 3.38 or less, which is slightly better than the other top-performing configurations. The maximum makespan ratio (not shown in Table 4.3) is 7.36 for HLF, whereas, for the SML heuristic, no workflow was observed to have a makespan ratio above 4.91. However, since the maximum makespan ratio is very sensitive to the workflows selected for evaluation, it is not further considered here.

Predictor	Priority	Failures	k	Mean MSR	99% MSR	**Mean MAQ**	1% MAQ
LR 80	LFF	Ignore	1	2.399	3.903	57.4%	47.2%
LR 80	Random	Persevere	1	2.444	4.217	57.4%	46.8%
LR 80	LIFO	Persevere	1	2.451	4.207	57.3%	46.9%

Table 4.4.: Top 3 configurations with respect to average memory allocation quality. 99% MSR denotes the 99th percentile of makespan ratio across all workflows. 1% MAQ denotes the long tail performance with respect to memory allocation quality.

Table 4.4 shows the best three configurations concerning average memory allocation quality. The best performing task priorities are LFF, Random, and LIFO, all using small backfilling parameters. Despite the high correlation between memory allocation quality and makespan ratio, the configurations optimizing average memory allocation quality are not the configurations that yield the best average makespan ratios, indicating the existence of a small trade-off between both performance criteria. However, average memory allocation qualities are at most 0.7 percentage points higher than those achieved by makespan ratio optimizing configurations. In addition, the small gain implies significantly worse average makespan ratios of up to 2.45 compared to 2.16 for the best makespan ratio optimizing configuration.

Overall, the configurations applying user estimates as prediction models do not perform well. The worst configurations all use the Power 2 user estimate prediction model, forming a separate cluster of configurations in Figure 4.3. The mean memory allocation quality of these configurations is constant because the prediction models are not updated at run time and are thus not affected by task execution order, i. e., scheduling. The top configurations all use the LR 80 prediction model. Out of the 19 configurations that are among the top 5% regarding both average makespan ratio and average memory allocation quality, 13 use the LR 80 prediction model, and 6 use the LR 95 prediction model.

Figure 4.3 also shows the *Pareto optimal* configurations with respect to low average makespan ratio, high average memory allocation quality, and low 99th percentile makespan ratio, i. e., the configurations for which no other configuration is better with respect to all three criteria. The Pareto front shows that a few configurations offer lower 99th percentile makespan ratios at the price of both average memory allocation quality and average makespan ratio. However, the top-performing configurations concerning average makespan ratio and average memory allocation quality also happen to be among the top-performing configurations with respect 99th percentile makespan ratio. For comparison, the 99th percentile makespan ratio ranges from 3.348 to 7.354 across all configurations.

Parameter Impact

Figure 4.4 shows the parameters that have the biggest impact on makespan ratio. For each parameter, the makespan ratios of the simulations using a given value for a given parameter are averaged. For instance, simulations using the LR 80 prediction model achieve a makespan ratio of 2.35 when averaging over all other parameters, i. e., basic task priority, failure handling strategy, backfilling parameter, workflow type, workflow instance, and time

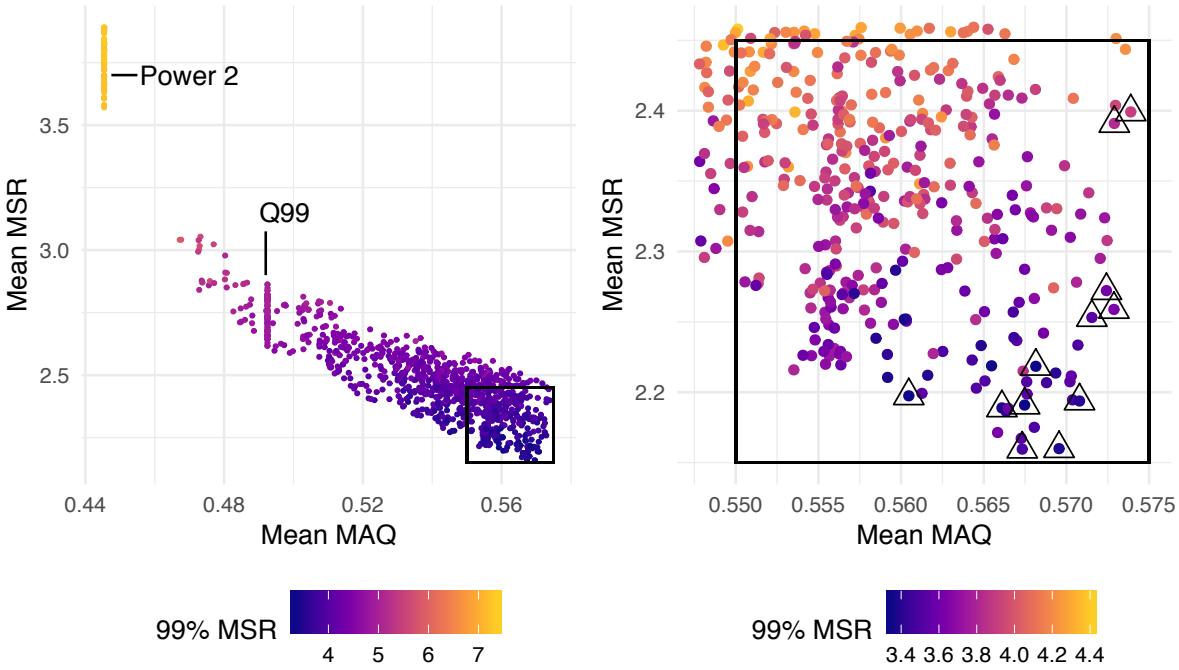

Figure 4.3.: Mean makespan ratio and memory allocation quality achieved by the 1134 configurations. The right panel is a close up of the area highlighted in the left panel, containing the best configurations. The configurations marked with triangles are Pareto optimal.

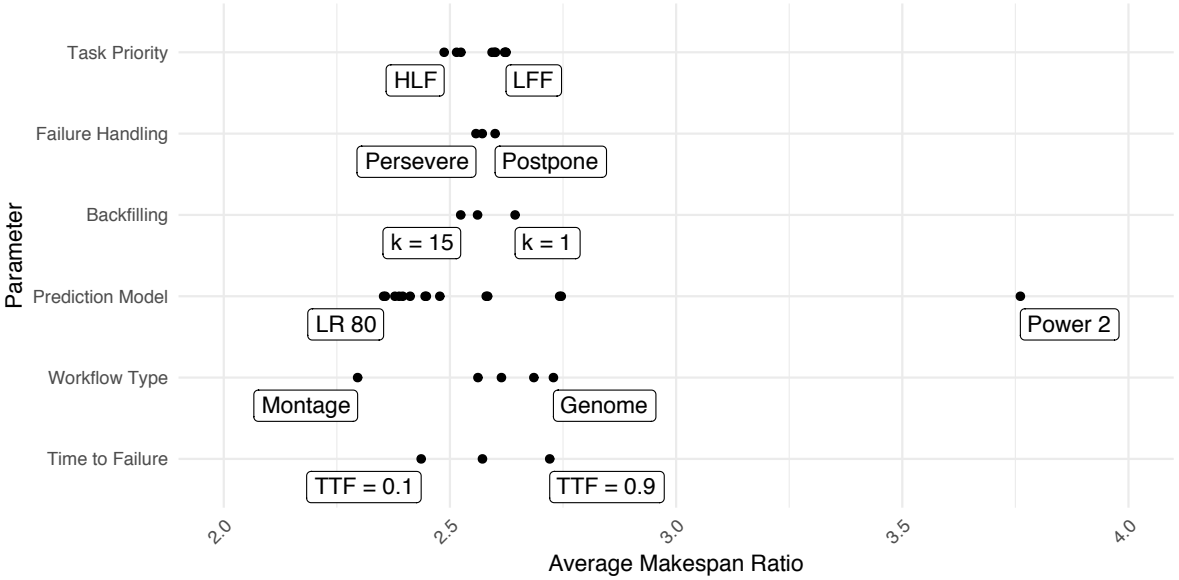

Figure 4.4.: Parameter impact on makespan ratio, as measured by averaging the makespan ratios of all simulations using a specific value for a specific parameter. The best and worst values for each parameter are labelled.

to failure. In contrast, the average makespan ratio of simulations that use the Power 2 prediction model is only 3.76, corresponding to a 60% increase in execution time. In general, comparing the worst choice for a parameter to the best choice gives a measure of the impact of a parameter on performance. By this measure, the prediction model is the most important parameter concerning makespan. Among the scheduling parameters, task priority has the biggest impact, followed by backfilling parameter.

Figure 4.4 also shows that the parameters workflow type and time to failure have a strong impact on average performance. This indicates that system performance depends considerably on the workflow, and averaging over all workflow types may hide potentials for performance optimization. For the memory allocation quality, a similar picture emerges, as shown in supplemental Figure B.3. The next section discusses the impact of different parameters for different workflow types and the potential of selecting the best scheduling heuristic and prediction model for a given workflow.

4.3.2. Workflow-Specific Configuration

In this section, the potential of optimizing the system configuration for a specific workflow is evaluated. This is particularly relevant for situations where the workflows executed by the system expose less variation than is present in the workflow corpus used in this evaluation. This corresponds to the setting where every workflow can be executed using a different configuration, e. g., to select the most appropriate prediction model or scheduling parameters for each workflow.

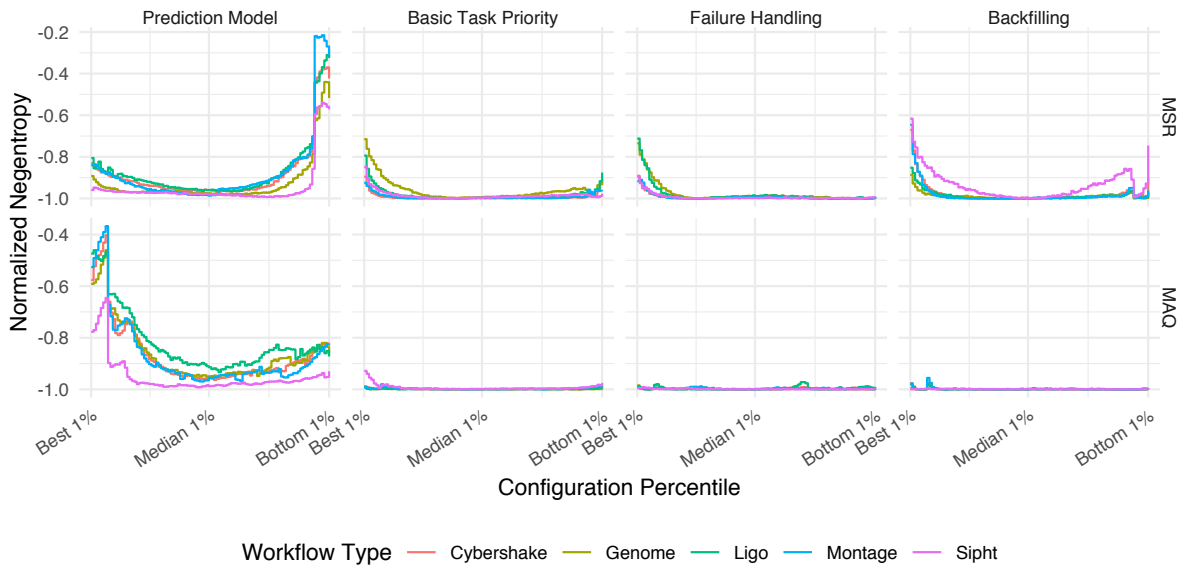

Figure 4.5.: Imbalance in value frequency for different parameters among the best 1%, second-best 1%, etc. configurations per workflow. Imbalance is measured as normalized negentropy. For a value of -1, every value is equally often present. For a value of 0, only one value is present.

Sensitivity Analysis

In this section, the frequency of parameter values in configurations as a function of configuration performance is analyzed. Intuitively, if all the top configurations use the same value of a parameter, e. g., the persevere failure handling strategy, using that value can be recommended. If, on the other hand, all failure handling strategies appear equally often among the top configurations either failure handling does not affect performance, or its effect depends on interactions with other parameters.

To analyze the sensitivity of the simulation with respect to its parameters, the degree to which a single parameter value dominates among configurations of a given quality is computed. The degree of domination is measured using normalized negative information entropy (negentropy). If all values are equally frequent, the normalized negentropy is -1. If only one value is present, the normalized negentropy is 0.

Let k be the number of different values for a parameter and let p_i be the frequency with which the i-th value is used in a set of simulations. The normalized negentropy is defined as

$$\text{Normalized Negentropy} = \sum_{i=1}^{k} p_i \log_k p_i \tag{4.10}$$

To determine the sensitivity of a parameter, the following procedure is applied.

1. The 1 134 executions of each workflow are partitioned into 100 sets according to their performance. The first set of the i-th workflow $S_{i,1}$ contains the best 1% configurations, the last set $S_{i,100}$ contains the bottom 1% configurations.

2. The frequency of the l-th value of the parameter is counted in the union of all top sets $p_{l,1} = |\{\text{execution uses } l\text{-th value of parameter} \mid \text{execution} \in \cup_{i=1}^{1500} S_{i,1}\}|/|\cup_{i=1}^{1500} S_{i,1}|$. The same is done for the union $\cup_{i=1}^{1500} S_{i,2}$ of the second-best sets to obtain $p_{l,2}$ and repeated up to the bottom sets to obtain $p_{l,100}$.

3. The normalized negentropy of value frequencies of the k parameter values $\{p^{j,i} \mid 1 \leq i \leq k\}$ is computed for all sets j according to Equation 4.10 ($j = 1$ for the top set, $j = 100$ for the bottom set).

To provide an example for the interpretation of the normalized negentropy, consider the top 1% configurations for the 300 Sipht workflows. The normalized negentropy for the backfilling parameter is -0.6 (top-right panel in Figure 4.5, pink curve), resulting from the following frequencies: Out of the workflow executions in the 300 top sets, 73% use a backfilling parameter of $k = 15$, which makes it the dominant choice. 24% of the simulations use $k = 5$, and only 3% use $k = 1$. This deviation from uniform frequencies indicates that backfilling has an impact on performance, although it does not indicate its magnitude.

Figure 4.5 shows the results for the four scheduling parameters task priority, predictor, failure handling, and backfilling with respect to MSR and MAQ. It is immediately apparent that performance is highly sensitive to the choice of the prediction model, indicated by a high negentropy. Simulation results mostly agree on which predictors achieve poor makespan ratio. For memory allocation quality, simulation results mostly agree on which prediction models deliver good performance, although not to the same degree.

Preferences for basic task priority, failure handling, and backfilling parameter are not as pronounced, but still strong for the makespan ratio criterion. Memory allocation quality is mostly indifferent to all but the prediction model parameter. The following subsections analyze the sensitive factors and show which parameter values achieve the best performance.

Top Performing Prediction Models

In the following, the best performing prediction models are analyzed by focusing on the top 5% configurations for each of the 1500 workflows ($S_{i,1}, \ldots, S_{i,5}$). This covers a total of $1134 \cdot 0.05 \cdot 1500 = 85\,050$ simulations. Figure 4.6 shows the frequency of each prediction model across the makespan ratio minimizing configurations (top) and the memory allocation quality maximizing configurations (bottom). The configurations are grouped by the time to failure of the workflow since that strongly influences the best performing prediction model. The higher the time to failure, the more conservative the best performing prediction models are, both for makespan ratio and memory allocation quality.

For a time to failure of 0.1, the most frequent prediction model across makespan ratio minimizing configurations is the LR 50 prediction model. This is considerably less conservative than the LR 80 prediction model, which is the best compromise for optimizing the makespan ratio averaged over all workflows. For a time to failure of 0.9, the best performing prediction models are LR 95 and LR 97. Notably, the most conservative models LR 99 and LR 100 are seldom amongst top configurations, not even when the time to failure is high. The PC predictors are generally less frequent in top configurations than the LR predictors but still

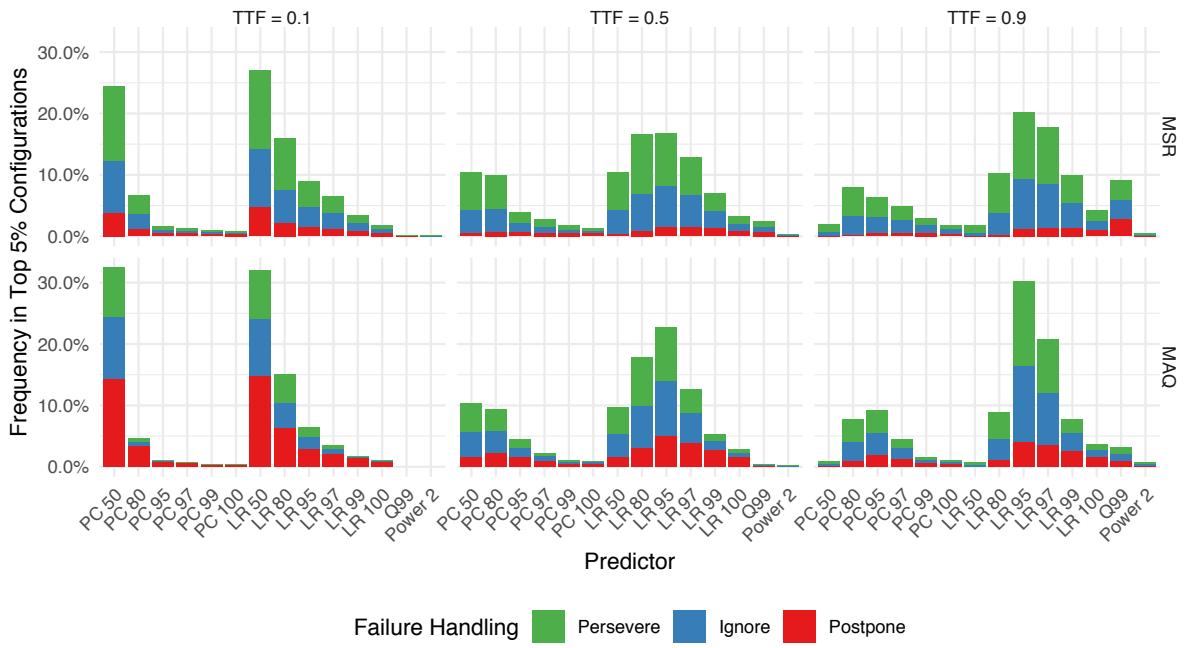

Figure 4.6.: The frequency of prediction models among the top 5% configurations with respect to makespan ratio and memory allocation quality. Colors indicate the fraction of configurations that use a specific failure handling strategy.

more frequent than user estimates. The prediction models that deliver best makespan ratios also tend to be the prediction models delivering the best memory allocation quality.

Overall, 49% of the makespan ratio minimizing configurations use the persevere strategy, 37% use the ignore strategy, and only 13% use the postpone strategy. When optimizing for memory allocation quality, the time to failure determines the most favorable failure handling strategy. For the scenarios with a time to failure of 0.1, 49% of top configurations use the postpone strategy, compared to only 20% for a time to failure of 0.9. This suggests that when the time to failure is short, one should favor optimistic schedulers, i. e., when a task fails, the scheduler tries another task. The reason is that when the cost of a failed attempt is small, a scheduler may try every ready task once to find those succeeding on their first attempt. When resource wastage per failed attempt is high, training data can be generated more efficiently by first focusing on completing a small set of tasks to improve the predictions for the remaining tasks.

Top Performing Scheduling Heuristics

Figure 4.7 shows the frequency of the basic task priorities in the top 5% configurations for each workflow. Unlike for prediction models, schedulers that optimize MAQ do not necessarily also optimize MSR. Across the configurations optimizing makespan ratio, there is a strong preference for the HLF, FIFO, and SML priorities, which are used in 58% of the top configurations. The most frequent is the HLF priority, which is used by 23% of the top

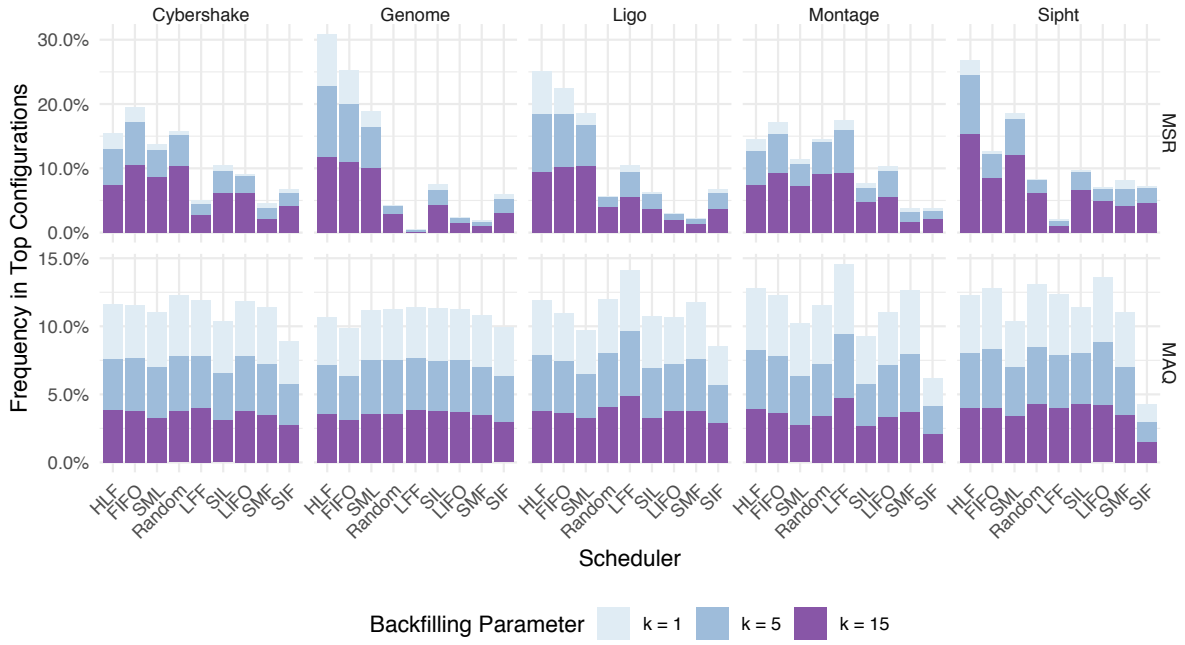

Figure 4.7.: The frequency of basic task priorities among the top 5% configurations with respect to makespan ratio and memory allocation quality. Colors indicate the fraction of configurations that use a specific backfilling parameter k.

configurations, 19% use the FIFO priority, and 16% use the SML priority. When optimizing memory allocation quality, there is a less clear preference for basic task priorities, and the preferred heuristics are not the same. 13% of the configurations use the LFF priority, 12% use the Random priority, and 12% use the HLF priority.

The best-performing schedulers also depend on the workflow type. As shown in Figure 4.5, the sensitivity of different workflow types to the scheduling parameters differs, e. g., Genome workflows are especially sensitive to the choice of basic task priority. As Figure 4.7 shows, the HLF heuristic is the most frequent for Genome, Ligo, and Sipht workflows. Montage workflows are fastest executed with the FIFO or LFF heuristic. Random task priorities perform better than perhaps expected. For Cybershake and Montage workflows, they are frequently among top configurations, however, not so for Genome, Ligo, and Sipht workflows. Memory allocation quality is mostly indifferent to basic task priority with the exception SIF, which rarely appears in top configurations for Montage and Sipht workflows. A possible explanation is that presenting tasks in ascending order of input size tends to produce more underpredictions, as the memory usages in the training set tend to be smaller than the memory usages of the tasks for which predictions are made. It is, however, not entirely clear why this effect is more pronounced on Montage and Sipht workflows.

With regard to the backfilling parameter, 55% of the makespan ratio optimizing configurations use the largest value, $k = 15$. However, as Figure 4.7 shows, this also depends on the basic task priority. For the HLF priority, there is a natural trade-off between strictly prioritizing tasks with more downstream work ($k = 1$) and increasing resource utilization

($k = 15$). For other heuristics, like random task priorities, strictly enforcing heuristics does not make sense, which is reflected by higher frequencies of larger backfilling parameters. For instance, 96.3% of the configurations using random priorities have $k > 1$. In contrast, when optimizing for memory allocation quality, all backfilling parameters are approximately equally frequent in the top configurations. Supplemental Figure B.4 shows the sensitivity of selected scheduling heuristics to backfilling using the negentropy approach.

Comparison to Static Configuration

In this section, the performance gains from choosing the best performing configuration for each workflow are evaluated. As a baseline, the configuration that yields the best average makespan ratio is chosen (LR 80, HLF, Persevere, $k = 5$, Table 4.3). This configuration represents the best compromise to optimize the makespan ratio of all workflows. The single best configuration per workflow concerning the makespan ratio is selected, and the resulting performance is compared to the performance under the compromise configuration. Figure 4.8 shows the change in makespan ratio and memory allocation quality. The makespan ratio gain denotes the ratio between the makespans delivered by the best and the compromise configuration, respectively. Since for each workflow, the configuration delivering the best makespan ratio is selected, the MSR gain is ≤ 1, i. e., the MSR is always at least as good as the MSR delivered by the compromise configuration. The memory allocation gain denotes the difference in memory allocation quality using the makespan ratio optimizing configuration and the compromise configuration and is specified in percentage points. Since the per-workflow configurations are selected to minimize the makespan ratio, memory allocation gains can be negative, although this is rarely the case.

Figure 4.8 shows that, on average, makespans can be reduced by approximately 10%. Memory allocation quality can be increased by 2.5 to 5 percentage points, depending on the workflow type. The Sipht and Genome workflows are most sensitive to configuration, in the extreme case finishing in 24% of the time needed by the compromise configuration.

4.4. Related Work

Most workflow scheduling research assumes that resource usage estimates, usually task execution duration and communication times, are given [Jennings and Stadler, 2014]. The problem of obtaining such estimates is considered a separate issue (see Section 2.4). On the other hand, research on predicting the resource usage of computational workloads usually focuses on sophisticated methods to learn from and extrapolate historical performance measurements, without considering the opportunity to create training data during the execution of a workflow. This chapter thus considers a new scenario where resource estimates change according to scheduling decisions.

Classical workflow scheduling algorithms [Kwok and Ahmad, 1999, Topcuoglu et al., 2002] use task execution and file transfer duration estimates to compute a schedule before execution. Here, memory constitutes the primary focus, motivated by memory-intensive workflows [Rheinländer et al., 2016, Bux et al., 2017] and the requirements of batch schedulers

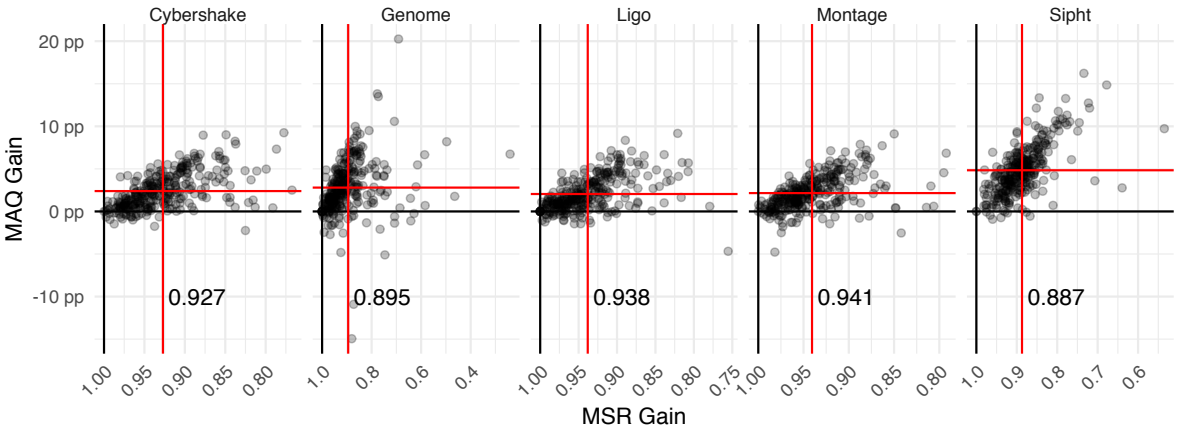

Figure 4.8.: Maximum makespan ratio gains from the dynamic configuration scenario. The resulting change in memory allocation quality (percentage points) is displayed on the vertical axis. The red lines mark the average gains.

and resource managers like YARN to specify the amount of main memory for each task before starting it. Dynamic scheduling algorithms place tasks at run time but typically do not plan and incorporate the entire dag into their decisions. Research on scheduling with both multi-variate resource demands and dependencies between tasks is still in its infancy [Grandl et al., 2016]. An overview of workflow scheduling algorithms is given in [Malawski et al., 2015, Rodriguez and Buyya, 2017], but none considers the online learning scenario covered here.

Since static schedulers depend on the accuracy of runtime estimates, prior studies have evaluated the scheduling quality under fixed degrees of prediction accuracy [Armstrong et al., 1998, Gibbons, 1997, Casanova et al., 2000, Tsafrir et al., 2007, Malawski et al., 2015]. This chapter focuses on the novel scenario where resource usage estimates and their accuracy are changing at run time in response to scheduling decisions. Silva et al. [Ferreira da Silva et al., 2015] showed that applying a task runtime prediction model during workflow execution yields improvements compared to estimating file sizes and run times in advance. However, in their work, the prediction model is built from prior executions before the workflow starts, and the actual intermediate result set sizes are passed to the pre-built model at run time. In contrast, this chapter focuses on the potential of continuously updating prediction models at run time.

Thamsen et al. developed the idea of collecting data during the execution of dataflows in a different context [Thamsen et al., 2016a]. For iterative dataflow graphs, i. e., cyclic directed graphs, resource usage measurements and statistics on dataset distributions can be collected between iterations and used to adapt resource allocations. This is used to adapt the degree of parallelism of operators to meet user-defined run-time targets. However, this assumes a white-box data model and malleable tasks, i. e., operators that can trade resources for run time. In [Thamsen et al., 2016b], an ensemble of parametric and nonparametric regression models is used to increase the robustness of scale-out predictions.

4.5. Discussion

This chapter evaluated the potential of learning task resource usage during the execution of a workflow. Compared to using user estimates throughout workflow execution, the proposed feedback-based scheduling offers the advantage of adapting to the actual resource usage of tasks during workflow execution. The results suggest that learning task resource usage with simple online prediction models can outperform user estimates. The proposed online approach also has the advantage that it allows to take into account input sizes of non-entry tasks, which are reliably available only during workflow execution.

It has become evident that the choice of prediction model has a huge impact on performance. The appropriate amount of conservatism depends on various factors, such as the time to failure, and thus the best prediction model varies from abstract task to abstract task. The next chapter proposes a method to optimize the degree of conservatism of a prediction model based on the resource usage characteristics of a set of tasks, which supersedes the prediction model selection approach taken in this chapter. Before proceeding to the next chapter, further research opportunities are discussed.

Real Memory Usages and User Estimates

The method has been evaluated using workflows with random peak memory usages and realistic topologies and run times. To this date, few publicly available real-world traces feature peak memory measurements, user estimates, and dependencies between tasks. However, at the time of writing, more and larger workflow traces from production environments become available [Versluis et al., 2019], some of which providing all of the required information. It would certainly be interesting to evaluate the proposed system on real-world workflows, which may show different characteristics, such as memory usage distributions with long tails. On the other hand, workload generators, as used in this work, have the advantage of creating a greater variety of workloads compared to data collected from individual use cases.

Dynamic Configuration

The results show that choosing the prediction model and scheduling heuristic according to workflow type and time to failure improves performance. It would be interesting to predict the best configuration prior to workflow execution. Yet, one of the advantages of feedback-based scheduling is incorporating information collected at run time. This approach suggests changing the scheduling heuristic or prediction model during the execution of the workflow. However, predicting the configuration that is most appropriate for the current part of the workflow seems challenging. First, there are many options to partition the workflow. Second, a workflow partition is a complex object, and its description features a lot of uncertainty, e. g., regarding input sizes, run times, and time to failure.

Adaptive Predictions

Based on the relative importance of tasks, a scheduler might adjust the degree of conservatism used in predicting the peak memory usage of a task. The ability to sacrifice resource

utilization to reduce the chance of task failure would enable a more fine-grained trade-off between memory allocation quality and execution time. For instance, it seems reasonable to assign more memory to tasks that have many children or need to finish before workflow execution can progress, e. g., to pass a synchronization barrier. The main challenge here is that informed choices require information that is difficult to predict, such as a task's importance for making progress in the workflow and the probability of failure as a function of allocated memory.

To implement a feedback-based approach in practice, further work on how to deal with user estimates seems necessary. Currently, the workflow management replaces user estimates as early as possible. Cross-validation may be desirable to determine when learned estimates are good enough to replace user estimates. It seems advisable to add user estimates to an ensemble of prediction models to increase precision and robustness of predictions.

Model Variations

The batch execution model considered here is only one of many choices to model workflow execution. For instance, it assumes that tasks can run until completion when given enough memory. Depending on the execution environment, estimates of maximum execution time may also be required. Always requesting the maximum allowed value can be used as a workaround, but it may have adverse effects on task queue times, i. e., the time until the resource manager serves the task. The obvious solution is to learn task run times as well. However, measuring the cost of overestimating run times is challenging because of the impact of a resource manager's policy. Resource managers are complex pieces of software, and their decisions may depend on the current load of the system, the requested resources, and other variables, which makes it hard to predict the consequences of requesting specific amounts of resources.

Similarly, several more model variations are thinkable. The batch execution model assumes that a resource manager kills tasks with insufficient memory allocations. However, tasks may be able to complete with lesser memory than requested, at the price of a runtime penalty for using swap. The scheduling literature refers to such trade-offs between resources as malleable resource demands. This adds considerable complexity to the scheduling problem, but also provides additional degrees of freedom for the scheduler to achieve its goals.

Qualitative Evaluation

Regarding the analysis of simulation results, a qualitative analysis of the strengths and weaknesses of different scheduling heuristics would be interesting. Qualitative analysis entails the question of why a scheduling heuristic performs well, in contrast to asking how well it performs in different scenarios. Consider the difficulties in quantitative analysis, e. g., approximating the optimal solution to a problem with lower bounds. Pinpointing problematic scheduling decisions is even harder because the quality of a scheduling decision cannot be assessed individually, but only in conjunction with previous and future scheduling decisions. However, simulating alternative traces by changing specific scheduling decisions might provide hints on the importance and quality of scheduling decisions in a simulation.

5. Low-Wastage Regression for Peak Memory Prediction

When executing scientific workflows on resources that require reservation, estimates of peak memory usage per task are needed. For instance, batch schedulers require peak memory estimates before execution and terminate tasks that exceed their requested memory. State-of-the-art workflow management systems still delegate the task of predicting peak memory usage to users who then find themselves in a dilemma: Allocating too much memory decreases throughput; allocating too little resources leads to task failures. Chapter 4 has shown that learning peak memory usage during the execution of a workflow is feasible. However, which prediction model performs best depends on various factors, such as the costs of task failures due to insufficient resources.

In this scenario, a prediction model that is aware of the asymmetric costs of prediction errors is desirable. Cost is here defined as the amount of unused resources, i. e., excess memory allocated to tasks over time. The prediction model should also take into account the costs of additional task execution attempts that follow task failures due to underprediction. Finally, the model should require little training data to be able to learn memory usage during the execution of a workflow.

This chapter formulates a novel objective function that quantifies the performance of a prediction model regarding the overall wasted allocated memory. This function takes into account follow-up costs of prediction errors that manifest in additional required attempts to execute a task. A method to optimize the objective is proposed and evaluated on real-world data from the IceCube high-energy physics experiment.

This chapter is structured as follows: Section 5.1 introduces the proposed machine learning approach. Section 5.2 provides background on production log files from the IceCube project, which have been used as a case study for evaluation. Section 5.3 covers the experimental setup, the baseline method, and the experimental results. Section 5.4 reviews prior work and Section 5.5 summarizes the chapter with a discussion and further research opportunities.

5.1. Method

This section proposes an approach for predicting peak memory usage for computational tasks. Section 5.1.1 introduces a novel asymmetric loss function to formalize a memory wastage minimization objective. Section 5.1.2 describes a rectified linear model for memory allocation. In Section 5.1.3, a solution to the non-convex optimization problem of choosing the model parameters is proposed. The method is based on quantile regression to generate initial solutions and refining them using iterative constrained optimization.

5.1.1. Memory Wastage Minimization

In this section, the memory wastage minimization problem is defined. The notation used is similar to the notation used in [Tovar et al., 2018]. Let D be a resource usage measurement set defining the run time τ_i, time to failure τ_i^*, peak memory usage r_i, and input size x_i for each of n tasks. Recall from Section 2.3.3 the equation for the wasted memory of an attempt to execute task i with a memory allocation of a.

$$\text{wastage}(\tau_i, \tau_i^*, r_i, a) = \begin{cases} (a - r_i)\tau_i & \text{if } a \geq r_i \\ a\tau_i^* & \text{otherwise} \end{cases}$$

The wasted memory depends on whether the allocated memory a is sufficient, i. e., whether it is greater or equal to the actual peak memory usage r_i. This definition of wastage reflects that within the batch execution model, a slight over-prediction is not a problem, but a slight under-prediction potentially wastes a lot of resources. The goal is to build a prediction model that accounts for this asymmetric loss and minimizes the wasted memory over all attempts needed to execute a set of tasks. This can be approached as a supervised learning problem where the training data is a resource usage measurement set.

The novelty of the proposed loss function is its awareness of exponential re-allocation. Since the goal is to minimize the memory wastage over all attempts needed to execute a set of tasks, the amount of memory allocated to the first attempt must take into account the costs of failures that might occur. Let a_{i1} denote the amount of memory allocated to the first attempt of the i-th task. Let b denote the factor that is used to increase the allocated resources after a failure due to insufficient resources. Using an exponential re-allocation strategy, the number of necessary attempts k_i for the i-th task is logarithmic in the relative underestimation r_i/a_{i1} of resource usage on the first attempt.

$$k_i = 1 + \max\left(0, \left\lceil \log_b \frac{r_i}{a_{i1}} \right\rceil\right) \tag{5.1}$$

The total resource wastage resulting from executing a set of tasks D with given first allocations a_{i1} can be split into oversizing and undersizing wastage. The undersizing wastage W_U equals the resources allocated to the $k_i - 1$ failed attempts. The oversizing wastage W_O equals the excess allocation in the successful final attempt.

$$W_U(D, a_{i1}, b) = \sum_{i=1}^{n} \sum_{j=1}^{k_i - 1} a_{i1} b^{j-1} \tau_i^* = \sum_{i=1}^{n} a_{i1} \frac{b^{k_i - 1} - 1}{b - 1} \tau_i^* \tag{5.2}$$

$$W_O(D, a_{i1}, b) = \sum_{i=1}^{n} (a_{i1} b^{k_i - 1} - r_i)\tau_i \tag{5.3}$$

$$W(D, a_{i1}, b) = W_U(D, a_{i1}, b) + W_O(D, a_{i1}, b) \tag{5.4}$$

The memory wastage minimization problem consists of finding first allocations a_{i1} and a basis b that minimize memory wastage for a given resource usage measurement set D. This is equivalent to maximizing memory allocation quality.

5.1.2. Rectified Linear Allocation Model

The solution proposed here is to compute the first allocation for the i-th task as a linear function of its input size x_i. To ensure positive allocation sizes, the allocation is rectified by capping it from below using a minimum allocation size $a_l > 0$. In case the first allocation is insufficient, exponential re-allocation using base b is applied. The memory allocated to the first attempt of the i-th task is:

$$a_{i1} = \max(\theta_1 x_i + \theta_0, a_l) \tag{5.5}$$

Substituting Equation 5.5 into Equation 5.4 yields the total wastage for a given resource usage measurement set D and parameters θ, b:

$$W(D, \theta, b) = \sum_{i=1}^{n} \underbrace{\left(\max(\theta_1 x_i + \theta_0, a_l) b^{k_i - 1} - r_i\right) \tau_i}_{\text{oversizing}} + \underbrace{\max(\theta_1 x_i + \theta_0, a_l) \frac{b^{k_i - 1} - 1}{b - 1} \tau_i^*}_{\text{undersizing}} \tag{5.6}$$

5.1.3. Constrained Optimization

This section demonstrates how to select the parameters of a rectified linear allocation model by finding a solution to a constrained, non-convex optimization problem with the multivariate objective function $W(D, \theta, b)$ subject to the constraint $b > 1$. The minimum allocation size a_l is considered to be fixed for simplicity, but it is straightforward to add it as another constrained parameter $a_l > 0$.

$$\underset{\theta \in \mathbb{R}^2, b \in \mathbb{R}}{\text{minimize}} \; W(D, \theta, b) \text{ s.t. } b > 1 \tag{5.7}$$

Solutions to the optimization problem stated in Equation 5.7 can be found using standard approaches such as grid search, evolutionary algorithms, or gradient descent. This chapter focuses on fast iterative optimization approaches to provide the possibility of training repeatedly during the workflow execution.

The optimization problem at hand is challenging because the objective function has many local optima. The reason for the many local optima is that the total wasted resources as a function of the first allocation follows a sawtooth-like pattern. Figure 5.1 shows an example of the memory allocation quality as a function of first allocation a_{i1} for a single task. Assume that the first allocation is too small. Note that increasing the first allocation *reduces* the memory allocation quality since it increases the wastage for every attempt until the point where one less failed attempt is required, and memory allocation quality increases abruptly. To a gradient descent method, this sawtooth pattern *always* suggests decreasing the first allocation, even if it is too small in the first place.

Since the total wastage during the execution of a set of tasks is the sum of the wastages per task, the objective function $W(D, \theta, b)$ is also non-convex. Figure 5.2 shows how the memory wastage $W(D, \theta, b)$ changes for an example data set D and different choices of base b, slope θ_1, and intercept θ_0. The underlying resource usage measurement set D is shown in supplemental Figure B.5.

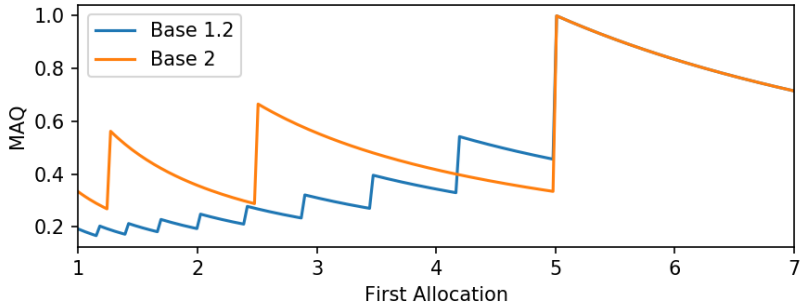

Figure 5.1.: Memory allocation quality (MAQ) as a function of the first allocation a_{i1} when using exponential re-allocation with different bases. The underlying resource usage measurement set contains a single task $D = (\tau_1 = 1, \tau_1^* = 1, r_1 = 5, x_1 = 0)$.

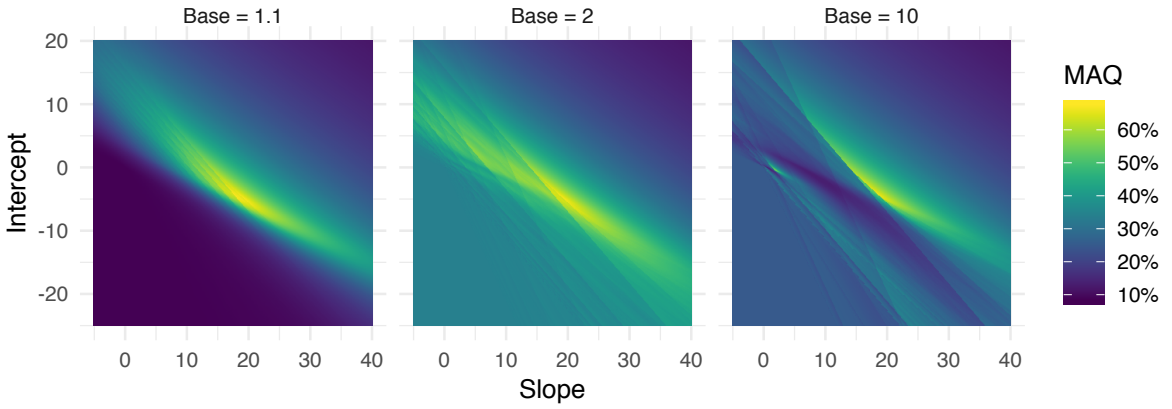

Figure 5.2.: Cross-sections of the objective function $W(D, \theta, b)$ for three values of b. It shows the memory allocation quality achieved by a rectified linear allocation model as a function of its parameters (slope θ_1 and intercept θ_0).

.

In the following, an optimization method that copes with the complex shape of the objective function and the constraint on the base parameter is presented. The basic approach is outlined in Algorithm 4. A fixed number k of initial slopes and intercepts is generated using quantile regression. Cobyla, a simplex-based constrained optimization method, is then used to refine these initial solutions iteratively. The two components, Cobyla and quantile regression, are described in the following.

Input: Resource usage measurements $D = (\tau, \tau^*, r, x)$
Result: Slope θ_1, intercept θ_0, and base b for computing the allocated peak memory according to Equation 5.5.

```
// Best value of the objective function so far
w_min ← ∞;
// Best model parameters so far
θ* ← (∞, ∞);
b* ← ∞;
// Quantiles used to find initial solutions
l ← interpolate k values linearly between 0.01 and √0.5;
for quantile q ∈ {1 − a² | a ∈ l} do
    // Regress q-th quantile of peak memory usage r on input size x
    θ1, θ0 ← Quantile Regression(q, x, r);
    // Refine initial slope and intercept starting from b = 2
    θ1, θ0, b ← Cobyla(θ1, θ0, 2);
    // Compute value of objective function
    w ← W(D, θ, b);
    // Replace best solution if outperformed
    if w < w_min then
        θ* ← (θ1, θ0);
        b* ← b;
        w_min ← w;
    end
end
return θ*, b*
```

Algorithm 4: The Low-Wastage Regression approach (LWR) applies the Cobyla method to iteratively refine initial solutions generated with quantile regression.

Cobyla

The optimization problem was approached using off-the-shelve optimization methods. In preliminary experiments, Constrained Optimization by Linear Approximation [Powell, 1994] delivered the best results without having to tune hyper-parameters. Constrained Optimization by Linear Approximation (Cobyla) is a simplex-based optimization method [Marazzi and Nocedal, 2002] for multivariate objective functions $f(x), x \in \mathbb{R}^n$. It iteratively replaces a point in a simplex, i. e., a set of $n+1$ points $x_i \in \mathbb{R}^n, 0 \leq i \leq n$, such that the simplex covers

successively smaller values of f. To replace a point, the values of f at the x_i are computed and linearly interpolated to generate a linear programming problem that approximates the objective function values and the parameter constraints. The basic idea is to replace one of the x_i in each iteration with the solution $\hat{x}$ to that linear programming problem until the solution converges, or a maximum number of function evaluations is reached. Additional heuristics affect the optimization process, for instance, to avoid degenerate simplices [Powell, 1994].The experiments were conducted using the implementation provided by the scikit-learn package [Pedregosa et al., 2011].

The method needs an initial solution and a maximum iteration count. The initial simplex is generated by perturbing the initial solution. The scale of the perturbation can be configured, but the default value of 1 was found to be appropriate for the scale of the data. The iteration limit is a means to trade-off between solution quality and speed. By default, the number of iterations is limited to 1000. However, a lower iteration limit of 100 was found to be sufficient for the problem at hand.

Quantile Regression for Initial Solutions

Due to the iterative refinement approach to optimization, the quality of Cobyla's solutions strongly depends on the initial solutions used (Section 5.3.4). This chapter proposes a heuristic to find initial solutions for a resource measurement set D based on quantile regression.

An intuitive approach to finding an initial solution is to use ordinary least squares regression to relate peak memory usages r_i to input sizes x_i. To generate a range of initial solutions, this approach was generalized to quantile regression. The intuition is that the number of tasks failing on their first attempt is a key parameter in the problem, as it determines the degree of conservatism (see also the prediction models in Chapter 4) of the prediction model. Quantile regression computes the slope and intercept of a line such that approximately a given percentage of the data is below that line. For instance, regressing the 95th-quantile of peak memory usage on the input size would yield a linear model that results in approximately 95% of the tasks succeeding on their first attempt. Note, however, that varying the quantile alone is not sufficient to minimize wastage, as it does not take into account task run times and costs resulting from follow up attempts.

To cover a range of scenarios, 10 values $l_1 \dots l_{10}$ are linearly interpolated between 0.01 and $\sqrt{0.5}$. Then, quantile regression lines for the quantiles $q \in \{1 - l_i^2\}$ are computed, and the corresponding slopes and intercepts are used as initial solutions. The quadratic interpolation was observed to result in more evenly spaced initial solutions in the slope-intercept space. In practice, it seldom seems advisable to have more than half of the tasks fail on their first attempt, hence the choice of the median as the smallest initial solution quantile. However, as this is only an initial solution, optimization can indeed converge at a solution that results in more than half of the tasks failing on their first attempt if this minimizes the resource wastage.

5.2. Case Study: IceCube Neutrino Observatory

LWR was evaluated using production logs from scientific workflows at the IceCube Neutrino Observatory. This section briefly introduces the IceCube project [Halzen, 2005], its scientific computing workloads, the IceProd workflow management system, and an analysis of the memory usage efficiency during ten months in the production system.

5.2.1. Scientific Workflows at IceCube

The IceCube Neutrino Observatory is a research facility located at the South Pole. It comprises 5160 optical sensors distributed over a volume of 1 km^3 of ice. The purpose of the setup is to detect and collect data about cosmic rays. It is, however, challenging to reconstruct such an event from the outputs of the optical sensors. This problem is solved by conducting large numbers of Monte Carlo simulations, simulating the sensor responses to hypothetical particles.

Simulations are divided into datasets, and each dataset consists of a number of tasks. Each task is an instance of an abstract task. Abstract tasks are defined by the program they invoke and the dataset they use. The invoked program is also referred to as the abstract task's name. Typical simulations comprise four abstract tasks that need to be executed in sequence. In the *generate* task, high-energy particles are propagated through the detector medium. Particle interactions trigger cascades in which single high-energy particles produce many lower-energy particles. The stochastic nature of the process also causes variance in peak memory usage. This task is followed by *hits*, in which the induced photons are tracked from their sources to the optical sensors. The *detector* task simulates the response of the detection electronics and produces the same kind of data as the real detector. Finally, *filter* applies the same event selection and reconstruction as the real system. Figure 5.3 outlines the structure of the simulation workflows.

IceProd Workflow Management System

Workflows are executed using IceProd [Schultz, 2015], a custom workflow management system running on top of HTCondor [Thain et al., 2005, Schultz et al., 2017]. Since HTCondor is a batch scheduler, it requires resource reservations (Section 2.3.2). An IceProd pilot is a job that runs in an HTCondor slot and claims tasks from a central manager that fit within the slot's allocation of CPUs, GPUs, main memory, storage space, and wall-clock time.

In IceProd, each task's initial resource requirements are based on a user estimate for the abstract task. In case of insufficient resources, a task is restarted with twice the requested resource whose constraint was violated, which corresponds to the exponential re-allocation approach with $b = 2$.

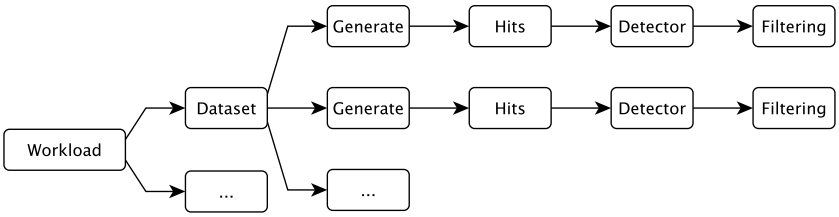

Figure 5.3.: Schematic of the IceCube workflow.

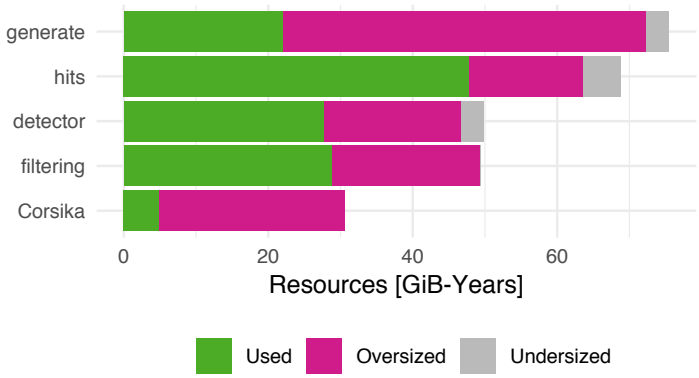

Figure 5.4.: Total resource allocation per task name. User estimates are rather conservative, strongly preferring oversizing compared to undersizing.

5.2.2. Production Log Data

This section analyzes the resource allocation and resource usage in the production system over ten months with respect to basic statistics, the quality of user estimates, and the predictability of peak memory usage.

The analysis covers 727 220 tasks grouped in 321 abstract tasks. The total memory usage amounts to 1.06 PiB, and the total CPU usage amounts to 76 Core-Years[1]. The total area, i. e., product of CPU time and memory usage, amounts to 133 GiB-Core-Years. The median task resource usage is 32 Core-Minutes and 1.4 GiB. Overall, tasks are compute-bound and memory-bound, and are not time-critical, as the workflows do not involve deadlines.

The total oversizing wastage amounts to 136 GiB-Years, whereas only 12 GiB-Years are allocated to tasks that failed due to insufficient memory. The time to failure tends to be uniformly distributed fraction of the time it would take the task to run to completion. Dividing the total 133 used GiB-Years by the 133+136+12 allocated GiB-Years, the overall MAQ is 47%.

Figure 5.4 shows the most resource consuming abstract tasks in the workload. Gray indicates memory allocated to failed tasks, pink indicates excessively allocated memory in successful tasks, and green indicates the amount of memory that was actually used. The combined width equals the total allocated resources. The much larger amount of oversizing

[1] 997 947 tasks with a total CPU usage of 136 Core-Years were excluded due to missing memory usage information.

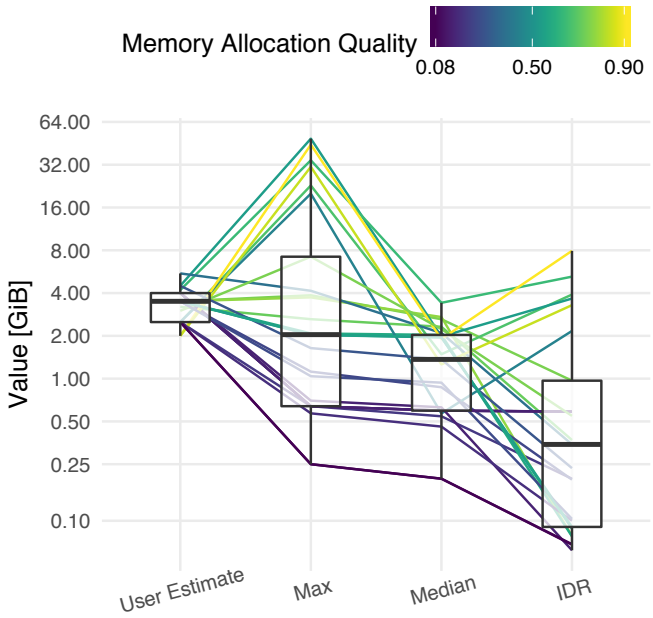

Figure 5.5.: Memory usage as estimated by IceProd users compared to actual peak, median, and interdecile memory usage of the 25 abstract tasks with the highest accumulated run time. Each line shows the four metrics for one abstract task.

wastage compared to undersizing wastage shows that user estimates are conservative, which confirms a known tendency of users to overestimate resource usage [Delimitrou and Kozyrakis, 2014].

Figure 5.5 shows the relationship between user estimates and actual memory usage for the 25 most time-consuming abstract tasks. These abstract tasks account for 73% of the total GiB-Year usage. It is striking that all user estimates are located between 2 and 6 GiB, whereas maximum peak memory usage ranges from 0.25 to 49 GiB. In addition, the variability of memory usage varies strongly. The interdecile range, i. e., the difference between the 90% percentile and 10% percentile can be as small as 0.06 GiB and as large as 7.93 GiB. Finally, memory allocation quality varies strongly across abstract tasks: The lowest MAQ is 8% and the highest MAQ is 91%. Interestingly, the tasks with the best MAQ are not necessarily the ones with the least variability. The rank correlation (Kendall) between MAQ and interdecile range is only 0.29.

In the workflows considered here, tasks consume at most one input file and produce one output file. Figure 5.6 shows the heterogeneity in memory usage and input file size as measured by the interdecile range. For a regression-based approach, the abstract tasks in the top-right corner are most interesting, since their variance in memory usage can potentially be explained by their input size. The correlation between input size and memory consumption was computed for each abstract task using the Pearson correlation coefficient between both variables, as displayed by the color of the points.

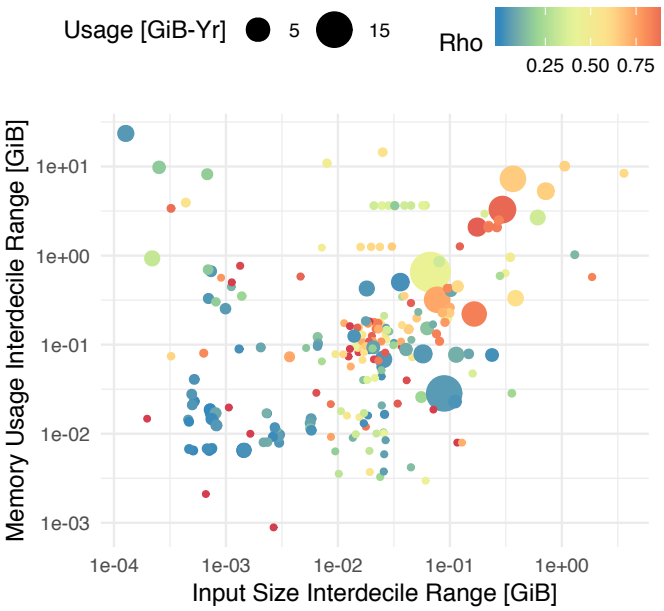

Figure 5.6.: Variability of input size and memory usage per abstract task. Regression-based memory allocation has the highest potential where input sizes and memory usage vary strongly (top-right corner) and are highly correlated (red).

The potential of other predictive features, such as the peak memory usage of a task's predecessor task was also evaluated. Although, in some cases, a high correlation (> 0.8) exists, only a small fraction of resources are allocated to tasks that are predictable in that sense. Thus, the feature was discarded as a predictor.

5.3. Experimental Results

This section covers the evaluation of the low-wastage regression (LWR) method and the baseline method [Tovar et al., 2018]. Both methods are evaluated in terms of memory allocation quality (see Section 2.3.3). A subsequent analysis evaluates the benefits of LWR's heuristic for generating initial solutions. Finally, LWR's model parameter choices on the IceCube data set are discussed.

5.3.1. Evaluation Approach

The proposed method is evaluated using the resource usage logs described in Section 5.2.2. Tasks are grouped by task name and data set identifier, e. g., all tasks of type detector in data set 20068. This grouping is also the granularity for which users have provided estimates of peak memory usage. The task groups are treated as resource measurement sets $D = (\tau, \tau^*, r, x)$. Each group is subsequently referred to as an abstract task and the user provided estimate of the abstract task's peak memory usage is referred to as u_i.

The production system maintains various metadata for every task, but for simplicity only the input file size has been used to predict peak memory usage. In addition to the run time r_i of each task, the log files also provide the start time and finish time of each task and the amount of memory u_i allocated to the first attempt of each task.

In the log files, only 1.6% of tasks ran out of memory, because user estimates are chosen very conservatively. Thus, τ_i^* is known for only a few tasks. However, on the available data, the relative times to failure τ_i^*/τ are approximately uniformly distributed between 0 and 1. In the evaluation, a relative time to failure of $\tau_i^*/\tau_i \in \{0.5, 1\}$ was chosen. Experiments in which the τ^* values have been sampled according to a uniform distribution $\mathcal{U}(0, 1)$ have also been conducted, but results varied only to a negligible extent and are thus not reported here.

The evaluation simulates an approach where memory is allocated according to user estimates u_i until the peak memory usage for a fraction of an abstract task has been measured. Then, the parameters θ, b are optimized on the collected training data to subsequently replace the user estimates by allocation sizes computed based on the metadata x_i. A fixed fraction $k \in \{0.05, 0.1, 0.5, 0.9\}$ of the tasks in an abstract task is used for training. The remaining data is used to evaluate memory allocation quality. An abstract task is divided into a training set and a test set based on the finish times of the tasks, using the first $k\%$ for training. To make sure the training data contains at least five jobs, only the abstract tasks were considered that comprise at least 100 jobs. This retains 99.7% of the tasks in the log, partitioned into 142 abstract tasks.

5.3.2. Baseline Method

Tovar et al. present an approach to the memory wastage minimization problem [Tovar et al., 2018]. The method uses a three-step re-allocation strategy: after a task has run out of memory for the first time, it is allocated the largest seen memory usage a_m of its abstract task so far. If the task runs out of memory again, the amount of memory a_v offered by the largest available compute node is allocated for the third attempt. It is assumed that no task requires more resources than the largest possible allocation size a_v. Let $D(\tau, \tau^*, r, x)$ be a resource usage measurement set, as defined in Section 2.3.3. The three-step re-allocation strategy f_m is defined as follows.

$$f_m(j) = \begin{cases} a_1 & j = 1 \\ a_m = \max\{r_i \in r\} & j = 2 \\ a_v & j = 3 \end{cases} \tag{5.8}$$

Compared to exponential re-allocation this is a more conservative strategy, because it needs at most three attempts for every task, given every task consumes at most a_v memory. However, determining a_v can be problematic in a pool of opportunistic compute nodes, as in the IceCube case study. In the experiments, a_v was thus set to the largest observed memory usage across all tasks in the log, which is as large as necessary and as small as possible. This is an optimistically low value, since the IceProd workflow management system uses variable size *job pilots* which are unlikely to exactly match the highest peak memory usage.

Let D be a resource measurement set, and let $P(r_i \leq r)$ denote the fraction of tasks that have a peak memory usage smaller or equal to r. In Tovar's method, the first allocation θ_0 is chosen such that it balances oversizing and undersizing wastage.

$$a_1 = \underset{\theta_0}{\text{argmin}}\ \theta_0 + a_m \sum_{r>\theta_0}^{a_m} P(r_i \leq r) \tag{5.9}$$

Intuitively, a large first allocation θ_0 reduces the number of failures. A small first allocation on the other hand increases the number of failures and thus the number of attempts for which a_m memory has to be allocated. The equation does not take into account a_v because, by definition, no task in D consumes more memory than a_m, and thus the third attempt with a_v resources is never necessary for the training data. The minimization problem can be solved elegantly using a single pass over the histogram of peak memory usages r and is thus in $\mathcal{O}(n)$. The run time of Tovar's implementation is actually pseudo-polynomial, because the number of bins in the histogram depends on the range of numeric values of the peak memory usages.

The method by Tovar et al. assigns each task the same amount of resources a_1, because task input sizes are not taken into account. This limits memory allocation quality in data sets where peak memory usage is heterogeneous and dependent on input size. Although Tovar et al. optimize for the same wastage criterion as the proposed LWR method, they do not take into account run times, which simplifies the computations. The method still performs very well in their experiments [Tovar et al., 2018]. Regarding the time to failure of a task, the pessimistic assumption $\tau_i^* = \tau_i$ is made. This means that even in the absence of input sizes, LWR has the potential to outperform the baseline method, because it takes into account run times and times to failure of tasks.

5.3.3. Memory Allocation Quality

In this section, the distribution of memory allocation quality across abstract tasks is reported. Abstract tasks are a natural unit of analysis since one prediction model is trained per abstract task. Finally, an aggregate analysis weights the memory allocation qualities by an abstract task's share of the overall allocated resources.

Abstract Tasks

Figure 5.7 shows the cumulative distribution of memory allocation quality across abstract tasks. User estimates score a median MAQ of 49%. LWR achieves 84% median MAQ using 5% of the data for training. The baseline method achieves a median MAQ of only 43% using the same amount of data for trainings. However, the baseline method improves to 55% and 83% when trained on 10% and 50% of the data, respectively.

Figure 5.8 shows the difference in memory allocation quality delivered by LWR and the baseline method, respectively. For some abstract tasks, LWR improves memory allocation quality by almost 75 percentage points. For some abstract tasks of type generate, memory

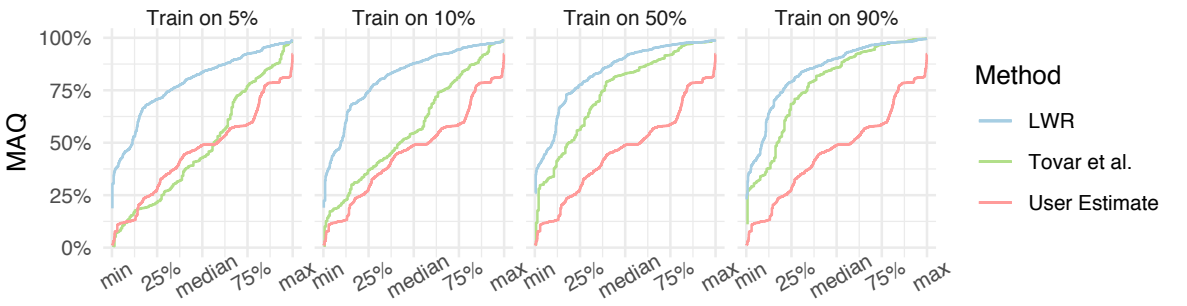

Figure 5.7.: Cumulative distributions of memory allocation quality (higher is better) for different prediction models. Memory allocation quality is computed per abstract task, where the first k% (with respect to a job's finish time in the logs) of the data are used for training and the rest for evaluation. The blue line shows the MAQs achieved when applying the peak memory usage estimates provided by the IceProd users.

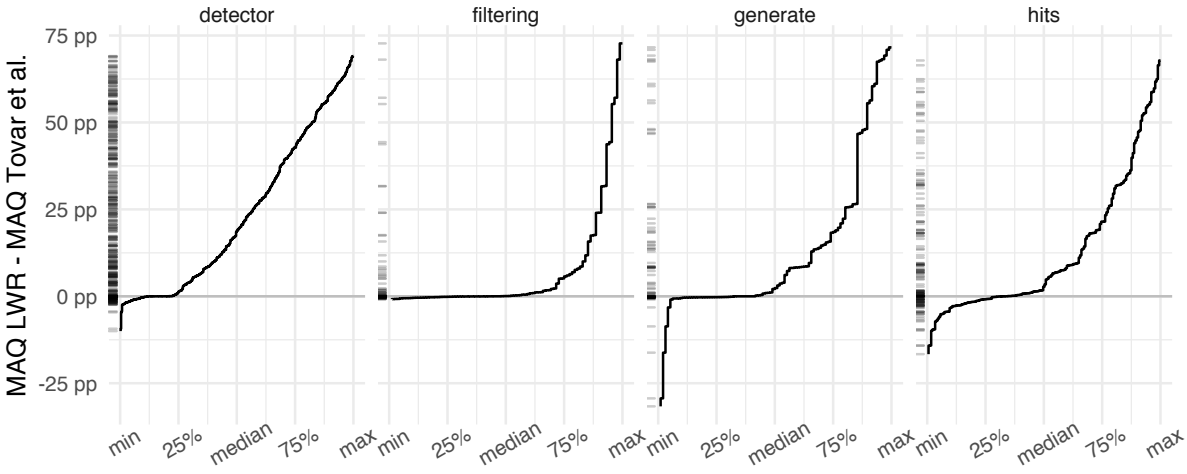

Figure 5.8.: Cumulative distribution of the differences in memory allocation quality (percentage points) when using LWR instead of the baseline per abstract task. For the vast majority of abstract tasks, LWR performs at least as good as the baseline. Subpar performance occurs on a few abstract tasks of type generate and hits.

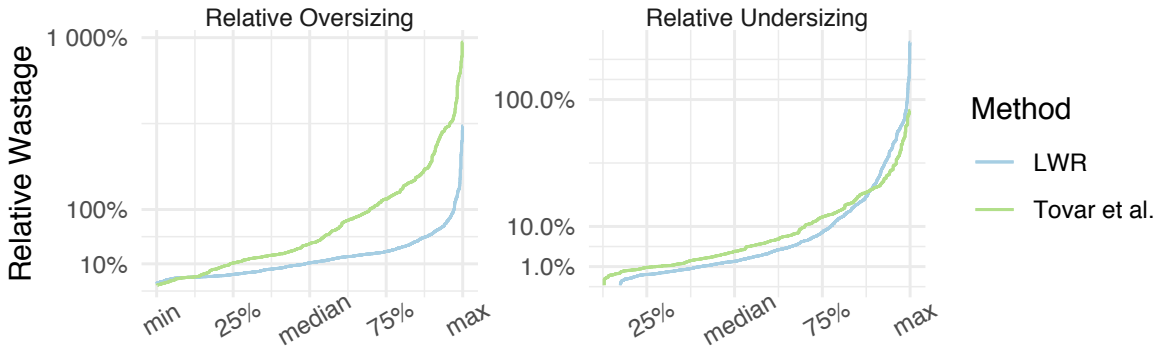

Figure 5.9.: Cumulative distribution of oversizing and undersizing wastage (lower is better), relative to the amount of used resources. This shows that the improvements of LWR stem mainly from reducing oversizing wastage.

allocation quality is up to 27 percentage points worse than for the baseline. Overall, LWR performs at least as good or much better than the baseline for most abstract tasks.

LWR Improvements

This section analyzes the source of LWR's improvements over the baseline method. The resource wastage resulting from prediction errors is split into oversizing and undersizing wastage. Let the relative oversizing denote the ratio between oversizing wastage W_O, as defined in Section 5.1.1, and usage $U(D)$. The total resource usage in a resource usage measurement set is defined as the sum of products of task run times and peak memory consumption.

$$U(D) = \sum_{i=1}^{n} r_i \tau_i \tag{5.10}$$

Let the relative undersizing wastage be defined analogously as $W_U/U(D)$. Figure 5.9 shows the cumulative distribution function of the relative oversizing and relative undersizing for LWR and the baseline method. This shows that LWR's reductions in MAQ stem mainly from reducing oversizing.

The potential to reduce oversizing wastage is largest in scenarios where input size correlates to peak memory usage. Here, a linear model can save significant amounts of resources by assigning less memory to jobs with smaller inputs and can avoid a substantial number of failures by allocating more resources to jobs with large inputs. In addition, the exponential re-allocation strategy better adapts to the actual peak memory usages than Tovar's maximum strategy.

Effective Memory Allocation Quality

In this section, the overall memory allocation quality is evaluated by weighting the memory allocation qualities of the individual abstract tasks by their share of the allocated resources.

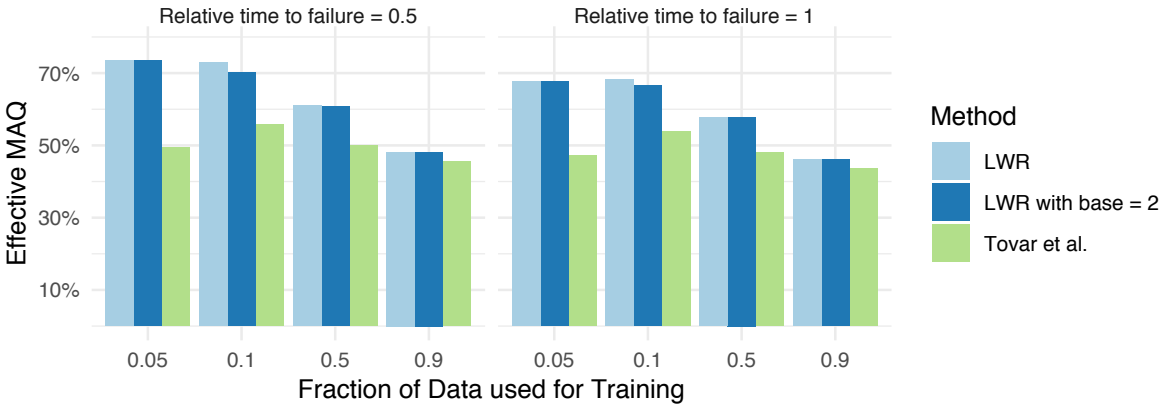

Figure 5.10.: Effective MAQ when using coarse grained user estimates during training and trained models afterwards. For reference, LWR with a fixed base of 2 is shown, which corresponds to the scenario where the re-allocation strategy is fixed.

In addition, the cost of training is taken into account by considering the resource wastage during the training phase. As long as the model has not been trained, the system has to rely on user estimates, which are typically less accurate than learned estimates.

In the log data, users have provided peak memory usage estimates for each of the 321 abstract tasks. Simulating a scenario where users invest less effort into estimating resource usage can be achieved by computing the median user estimate per task name (e. g., generate, hits, detector, etc.). This reduces the 321 user estimates to one user estimate for each of 16 task names.

Figure 5.10 shows the effective memory allocation quality that can be achieved when training models during workflow execution. When replacing user estimates with LWR's predictions after training on the first 5% of jobs of each abstract task, overall memory allocation quality can be improved to 71.2%. When using 90% of the data for training, the overall achievable memory allocation quality is roughly the same as completely relying on user estimates, since the learned models are applied only to 10% of the jobs.

5.3.4. Sensitivity to Initial Solutions

Cobyla is naturally susceptible to poor choices of initial solutions. Figure 5.11 shows an example of the solutions evaluated by Cobyla for different initial solutions. Cobyla has troubles to escape the region in the middle left of the figure (orange trajectory) because of the many local optima, which obscure the location of the global optimum. The lower left part of the figure contains solutions that would result in negative memory allocations. The rectification of the allocation function results in zero gradients in this area. This leads to early convergence (pink trajectory) and poor solutions as shown in Figure 5.12.

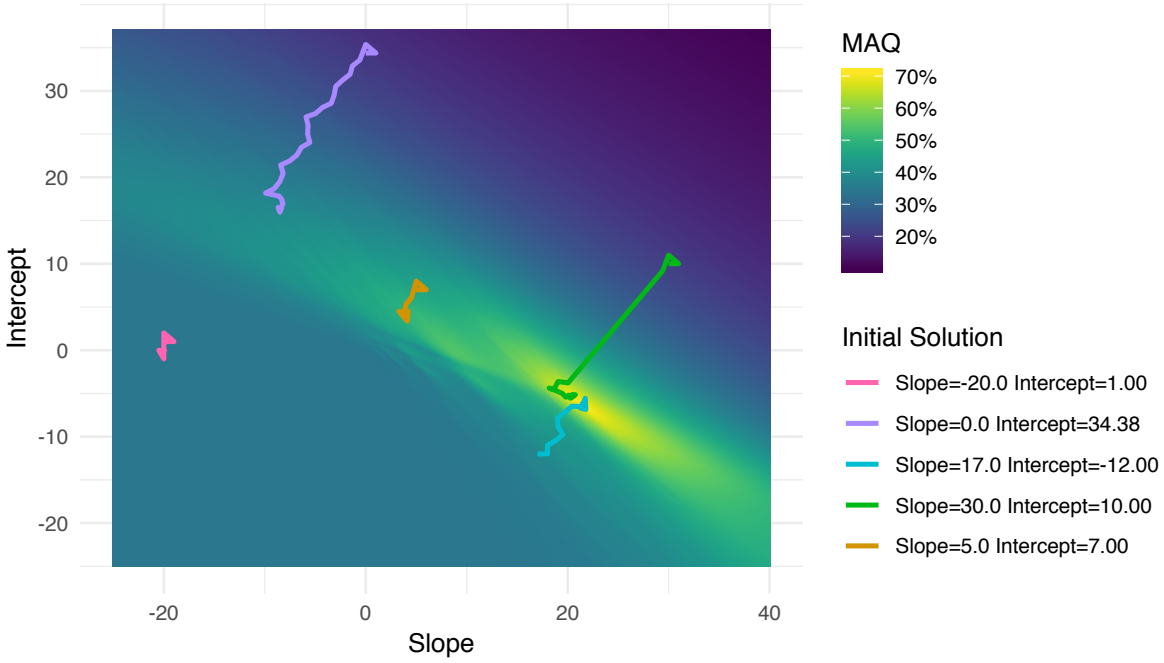

Figure 5.11.: Solutions evaluated by Cobyla, for different initial solutions. The quality of the final solution strongly depends on the starting point of the search.

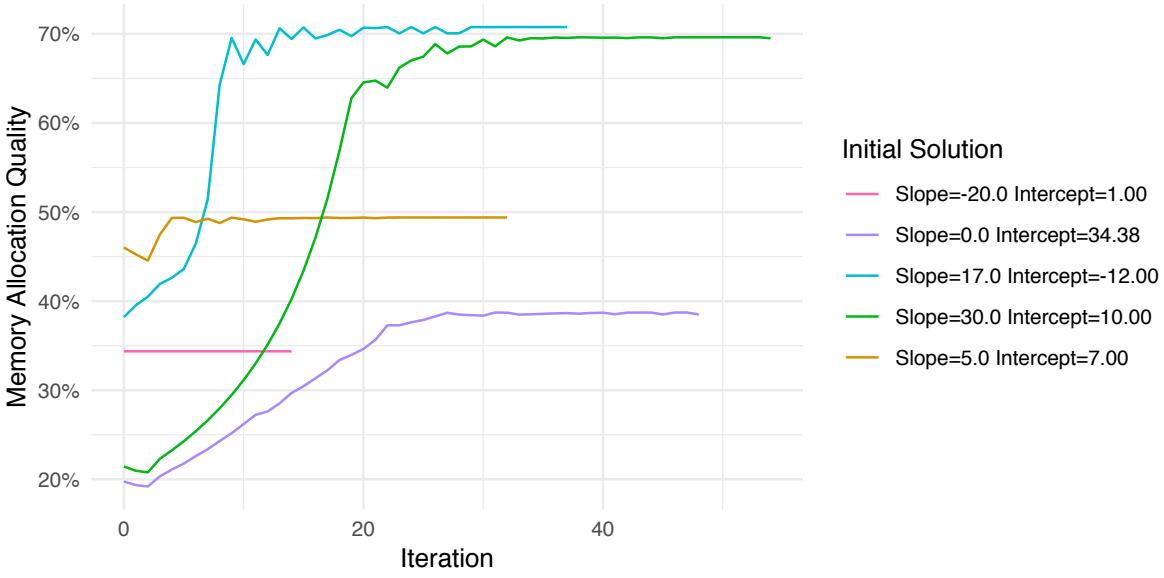

Figure 5.12.: Memory allocation quality for the solutions evaluated by Cobyla, as shown in Figure 5.11. The initial solution affects both the final solution quality as well as the number of iterations until convergence.

For the evaluation of LWR's heuristic for generating initial solutions (see Section 5.1.3), it has been compared to two other heuristics. This results in the following three methods for generating initial solutions:

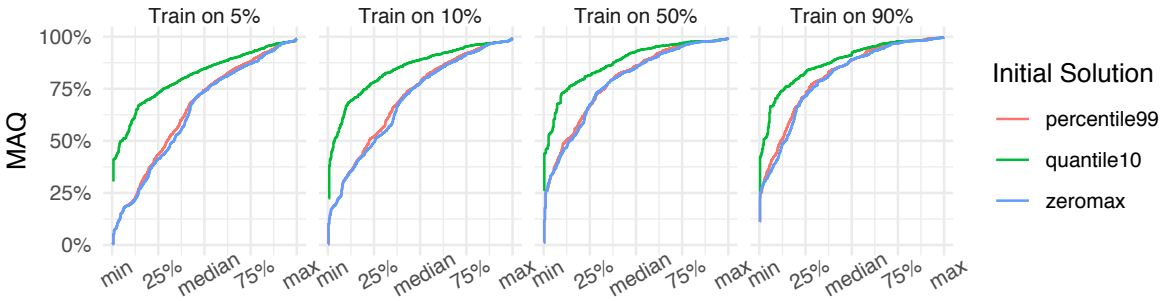

Figure 5.13.: Generating a range of initial solution using the `quantile10` heuristic outperforms heuristics that generate single initial solutions by a large margin.

- `zeromax`: initialize the slope $\theta_1 = 0$ and the intercept $\theta_0 = \max\{r_i\}$. This results in first allocations that allow every task in the training data set to succeed on its first attempt. This is a very conservative starting point. However, other initial solutions usually exist which are closer to the actual peak memory usages and still let all tasks succeed on their first attempt. Such solutions could be found by considering tangents to the upper part of the convex hull of the (x_i, r_i). The convex hull, however, is not robust with respect to outliers, and quantile regression provides a flexible alternative.

- `quantile10`: evaluate slopes and intercepts generated by ten quadratically spaced quantiles (see Section 5.1.3). This is the default method in LWR.

- `percentile99`: select θ by running a single quantile regression with a fixed quantile of $q = 0.99$. The idea is to validate whether a single quantile regression that produces a conservative slope θ_1 and intercept θ_0 suffices to find good model parameters.

Figure 5.13 shows that LWR's method (`quantile10`) outperforms the heuristics that only use a single initial solution (`zeromax` and `percentile99`) by a large margin, allowing LWR to achieve much higher memory allocation qualities.

5.3.5. Model Parameter Choices on the IceCube Data Set

Figure 5.14 shows the concrete slopes and intercepts chosen by LWR for different abstract tasks. This shows that similarities across abstract tasks with the same name exist. For instance, for abstract tasks of type generate, LWR mostly returns constant prediction models, i. e., selects a slope of zero. For abstract tasks of type detector, LWR commonly chooses slopes between zero and ten. The selected parameters appear to be robust and in reasonable ranges, except for very few abstract tasks of type hits. Manual inspection shows that these abstract tasks have almost constant input size. The implementation of quantile regression tries to catch degenerate slopes by falling back to simple quantiles when there is no variation in the input size. However, datasets occasionally expose extremely small variation, which leads to extremely large slopes and thus also to potentially very small intercepts. Figure 5.14

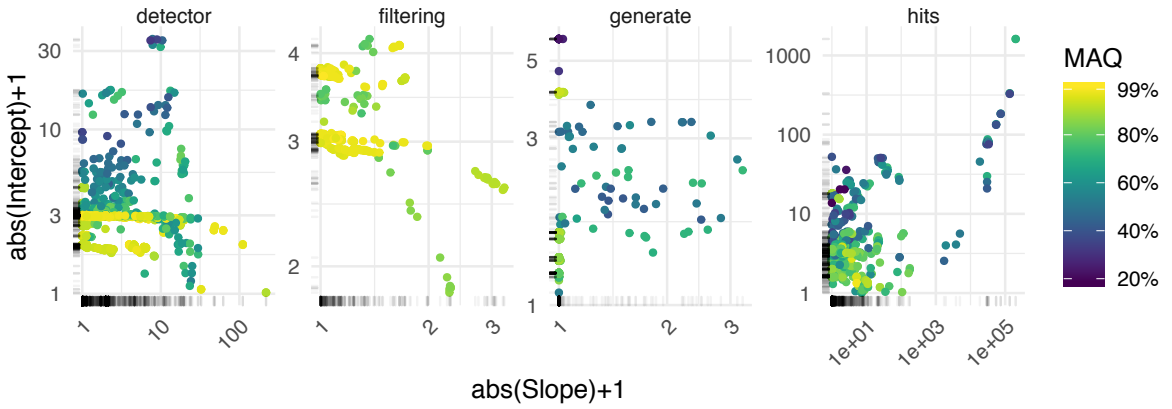

Figure 5.14.: Chosen slopes and intercepts for the abstract tasks with the most frequent task names. Values are mostly positive and moderate in magnitude. For better display of the outliers, the values have been log-transformed after taking their absolute value and adding one.

also shows that these extreme choices often have an adverse impact on memory allocation quality.

The initial choice of two for the base is maintained for 82% of the abstract tasks. The maximum base chosen across all abstract tasks was 3.82. In summary, changing the base can be beneficial in some cases, but doubling seems to be a sensible default. This is also apparent from Figure 5.10, which shows that fixing the base to $b = 2$ causes only a minor decrease in memory allocation quality.

5.4. Related Work

In Section 2.4, an overview of machine learning-based prediction methods for memory usage is provided. Of special relevance for this chapter are approaches using asymmetric objective functions.

The closest related research was conducted by Tovar et al. [Tovar et al., 2018], who also propose a strategy to solve the memory wastage minimization problem. They chose the first allocation such that the sum of over- and undersizing wastage is minimized (see Section 5.3.2). The assumption is that a job is restarted using a maximum allocation of a_m, should the first allocation be insufficient. The computation is simplified by assuming independence of run times and peak memory consumption. However, as shown in the evaluation, taking into account input sizes has a high potential of reducing memory wastage. Furthermore, Tovar et al. do not take into account run times, which can significantly affect the best model parameters, as shown in supplemental Figure B.5.

Gaussier et al. [Gaussier et al., 2015] proposed a machine learning approach with asymmetric loss function for predicting the run times of batch jobs. This is useful for batch schedulers with backfilling, i. e., that allow short jobs to skip the queue to fill currently idle resources that are insufficient for the job at the head of the queue [Tsafrir et al., 2007]. Similar to the

memory allocation problem, over- and underprediction have asymmetric effects on backfilling. However, maximum execution times behave fundamentally different, as excess time can be allocated to other jobs, while excess memory cannot.

Zhang et al. [Zhang et al., 2018] have proposed a clustering-based approach that determines groups of jobs with similar resource usage within a workflow. Each job is assigned to one of the clusters based on its measured resource usage from prior executions. When replacing user estimates with the maximal resource usage of the cluster of a job, resource wastage can be reduced. A disadvantage of this approach is that there is no means to predict the cluster of a job at runtime other than from a previous execution. In contrast, LWR takes into account additional information about tasks, such as input size to predict peak memory usage prior to the first execution of a job.

5.5. Discussion

In this chapter, a machine learning-based resource allocation method (LWR) has been proposed that minimizes expected memory wastage in scenarios that require resource reservations. An evaluation using 70 000 tasks from ten months of production log data has shown that automatically predicting peak memory usages can outperform user estimates by a large margin. Overall memory allocation quality could be improved from 47% to over 70% by replacing user estimates after measuring input sizes and memory usage for each abstract task for a small amount of time. If the saved resources could be translated completely to throughput gains, this would correspond to a 57% increase in throughput using the same amount of resources.

In the context of this thesis, LWR provides a foundation for learning the memory usage of tasks during the execution of a workflow. In the following, further optimization potentials and directions for future research are discussed.

Model Flexibility

Although a linear model performs well in the task of selecting first allocations based on input size in the IceCube case study, other data sets might require more flexible models. A straightforward approach is to fit a higher-order polynomial instead of a linear function. For strongly multimodal distributions, piecewise linear models may be more appropriate, e. g., decision trees. Decision trees or, more generally, random forests would also allow for incorporating additional predictive information, such as program versions and parameters. Random forests are robust with respect to feature selection and allow the use of categorical variables.

Adaptive Learning

In practice, the question of how often and when to train a prediction model arises. In the IceCube case study, models typically converged after having seen 10% of the overall available training data (results not shown). Thus, simply training and applying the model after fixed

fractions of training data have arrived, e. g., 1%, 5%, 10%, ... would be an option. However, a solution that adapts the frequency of training and determines minimal training set sizes might result in a more robust and more efficient solution.

The second degree of freedom is the choice of optimization method, which depends on various problem parameters. One central parameter is the time it takes to evaluate the objective function, which in turn heavily depends on the size of the resource usage measurement set used for training. Choosing the optimization method based on problem parameters might improve solution quality and speed. In addition, forming an ensemble of methods could improve results. As shown in Figure 5.8, the baseline method finds better first allocations on some data sets and could thus contribute to an ensemble.

Workflow Management System Integration

For applying the proposed prediction method in practice, a few problems have to be solved. To train models and use their prediction during the execution of the workflow, two options exist.

One option is to include LWR as a general mechanism into the workflow management system. The system then needs to be extended such that it collects and communicates input sizes, task run times, and peak memory usages to an implementation of the prediction model.

The other option is to embed the prediction in a concrete workflow by defining tasks that are responsible for training prediction models and querying them for predictions. However, this is not possible in all workflow management systems, as it requires the possibility to dynamically define peak memory requirements using values generated by the workflow. This is possible, for instance, in Nextflow [Di Tommaso et al., 2017], but accessing the peak memory usage measurements during the execution of a workflow is currently only possible through a workaround. The IceProd workflow management system, on the other hand, is written in Python, like LWR's prototypical implementation. Thus no major hurdles are expected regarding an extension of the system with automated predictions.

6. Conclusion

6.1. Summary

This section summarizes the contributions of the core chapters of this thesis.

Chapter 3 presents the LOS randomized search algorithm for scheduling. The method extends the list scheduling approach to the static task graph scheduling problem. The chapter proposes a level-based partitioning of the tasks in a workflow. L-Orders are introduced as a principled way to generate task priorities that result in a topological ordering of the tasks. The proposed method derives variations of topological orderings by shuffling the priorities of tasks in individual levels of the graph. The search process is driven by estimates of the probability of improving over a current solution when shuffling certain levels. The topological sorting of tasks allows for generating randomized variations of schedules computed by the well-cited HEFT heuristic [Topcuoglu et al., 2002]. The schedules found by LOS are often shorter (5% to 40%) than both the schedules produced by HEFT and the schedules produced by a baseline method [Zhao and Sakellariou, 2004].

Chapter 4 evaluates the potential of learning resource usage estimates during the execution of a workflow. A feedback-based workflow execution model is proposed that monitors and models task memory usage at run time. Different scheduling heuristics, e. g., the least finished first heuristic, are evaluated in combination with different prediction models, e. g., a conservative online version of linear regression. The simulations are conducted in a workflow simulator based on the CloudSim framework, extended with custom scheduling and online prediction methods. It was found that learning memory usage at run time outperforms fixed estimates of memory usage by a large margin, resulting in higher resource utilization (up to 15 percentage points) and lower workflow execution times (up to 76%).

Chapter 5 presents a machine learning model for estimating the peak memory usage of tasks in a workflow based on their input file sizes. A cost function is defined that relates prediction errors to memory wastage under exponential re-allocation. The problem of minimizing the costs of prediction errors is solved by combining a simplex-based optimization method with quantile regression to generate initial solutions. The method is evaluated using real-world workflows from the IceCube research project. This chapter focuses on developing a prediction model providing recommendations on how much memory to allocate to each task, but it does not consider scheduling. It was found that the memory allocation quality can be improved by replacing user estimates with predicted peak memory usages (from ca. 50% to 75%).

6.2. Challenges and Opportunities

This thesis concludes with a discussion of challenges and opportunities in four areas: Prediction accuracy, design of scheduling heuristics, qualitative performance insights, and trust and practicality issues of automated resource allocation.

6.2.1. Prediction Accuracy

A central topic for the practical application of automatic resource allocation is prediction accuracy. Chapter 4 and Chapter 5 have shown that learned estimates can be much more accurate than user estimates.

The models proposed in this thesis achieve high performance using only input sizes. Nevertheless, the predictive power of input sizes may be limited in different data sets. To be able to use the same model in a wide range of scenarios, the model should be able to take into account a wide range of features and select the most appropriate features automatically. As discussed in Section 2.4, several models in the related work also take into account categorical attributes, such as the user's identity. Applying the LWR method within the leaves of a decision tree seems promising to improve its versatility and accuracy.

Prior Information

Although Chapter 5 has shown that as little as 5% of the data can suffice to learn estimates without prior knowledge, incorporating prior information could help to increase prediction accuracy further.

Historical measurements could be used to build hypotheses about the relationship between resource usage and input size. Hypotheses could concern functional form only, e. g., constant, linear, or quadratic, or concrete parameters as well, e. g., slope and intercept. For instance, previously fit model parameters could serve as a prior distribution on model parameters of similar abstract tasks, similar to the analysis of model parameter similarities in Chapter 5.

Online Features

In addition to prior information, more data could be collected during the execution of the workflow:

- Measurements from failed task attempts provide lower bounds on peak memory usage. Including this type of censored data [Klein and Moeschberger, 1997] in a prediction model may improve early predictions.

- LWR predicts peak memory usage based on finished tasks of the same abstract task. In the workflow graph, these tasks are often siblings. This approach can be generalized by relating a task's resource usage to the resource usage of its ancestors. Ancestor-based prediction could either happen at the abstract task level (similar to analytical benchmarking [Iverson et al., 1999]) or the file level. In the latter case, the task would be to infer the latent properties of a file by observing the resources needed to process

its contents. In the IceCube case study, some predictive potential was found for such features, see Section 5.2.2.

6.2.2. Heuristics Design

The range of possible scheduling scenarios is as large as the heterogeneity of the needs of specific applications. In this thesis, the main goal is to mitigate main memory bottlenecks. However, there is no universal scheduling solution that accounts for all kinds of performance-critical aspects, which necessitates the ongoing development of heuristics. The field of heuristics design poses research challenges in the following areas.

High-Fidelity Simulation and Traces

Tools and data for simulation are an important prerequisite to develop and test heuristics. The accuracy and efficiency of the simulation are essential; thus, the development of workflow simulators is an ongoing research area. Examples include WorkflowSim [Chen and Deelman, 2012], DynamicCloudSim [Bux and Leser, 2015], and recently Wrench [Casanova et al., 2019]. However, a simulation framework is necessary but not sufficient for research on scheduling and prediction. Realistic workloads and execution traces are also essential to evaluate the performance of solutions under realistic conditions. At the time of writing, more and larger traces become publicly available. A recent example is the workflow trace archive [Versluis et al., 2019] that gathers new traces and previously published traces [Reiss et al., 2012] in a common format.

Exploring the Boundaries between Scheduling and Prediction

This thesis combines scheduling and prediction in two different ways. In Chapter 3, statistical methods are used to predict where to search for improvements of a schedule. In Chapter 4 and Chapter 5, predictions are used to inform a scheduler about the expected resource usage of a task.

Machine learning could also be used to predict the quality of scheduling decisions. For instance, scheduling with reinforcement learning has recently received an increased amount of interest [Mao et al., 2019, Moghadam and Babamir, 2018, Thamsen et al., 2017, Chen et al., 2017]. Task priorities generated by list scheduling heuristics can also be understood as predictions. An interesting combination would be to predict a task's importance using machine learning. Both ideas are examples of learned heuristics that strongly rely on simulation to generate sufficient data from which to learn.

Evaluation Methods

In addition to research on new heuristics and new methods to learn heuristics, research on evaluating heuristics is necessary.

Lower bounds on workflow execution time for more complex dags are needed to assess the quality of a solution produced by a heuristic. The lower bounds used in this thesis have been

used for decades [Jain and Rajaraman, 1994]. However, tight lower bounds for task graphs on heterogeneous infrastructures [Atef et al., 2017] and with memory requirements [Im et al., 2015, Song et al., 2019] are an ongoing research topic.

Another challenge in evaluating scheduling heuristics is the choice of workload. Ideally, the heuristic is evaluated on the workload it is designed to schedule. However, workloads are difficult to characterize comprehensively and also change over time. For determining the impact of the chosen workflows on the results, a wide variety of workflow generators [Cordeiro et al., 2010, Ferreira da Silva et al., 2014, Gupta et al., 2017] should be used.

An alternative approach is to integrate workflow synthesis and scheduler evaluation. An optimization method could try to repeatedly modify workflows in a way that creates particularly low performance, remotely related to the idea of generative adversarial nets [Goodfellow et al., 2014]. Evolutionary algorithms seem appropriate for this task. A coupled synthesis-evaluation framework could be used for a variety of tasks, such as finding weaknesses particular to a scheduling heuristic or generating workflows that are difficult for all considered heuristics, e. g., as a benchmark workflow corpus.

6.2.3. Qualitative Research

Chapter 4 applies quantitative measures to determine which scheduling heuristics and prediction models perform best under which circumstances. To complement these results, research tools and methods for gaining qualitative insights would be desirable. Challenges are the extraction of salient patterns from large sets of simulation traces, the comparison of workflows, and the comparison of schedules.

Extracting Salient Patterns from Trace Sets

Simulations are capable of generating large numbers of workflow execution traces. Selecting traces that expose interesting behavior is a challenge. One approach is to look for particularly low or high values in quantitative measures, e. g., memory allocation quality and makespan. However, even a reduced-size set of traces may not be very helpful in itself. For instance, researchers are likely interested in whether poor performance can be attributed to a single flaw in a scheduling heuristic or whether there are multiple problematic behaviors.

Unsupervised learning methods could be used to group similar traces. However, defining similarity measures on complex mathematical objects like schedules or workflows is challenging. Representation learning and graph embeddings may be used for clustering [Hamilton et al., 2017, Cai et al., 2018].

Manual Comparison of Workflows and Traces

A central challenge for qualitative research is the ability to compare simulation inputs, i. e., workflows, and simulation outputs, i. e., simulation traces. However, tools and methods for manual inspection of inputs and outputs are lacking. In static task graph scheduling, the input instances are graphs with multi-dimensional annotations. Drawing large graphs is a

difficult problem in itself [Herman et al., 2000], which is further complicated when having to visualize the complex interplay with the execution environment.

Comparing execution traces is similarly challenging. Traces may contain thousands of events, e.g., tasks become ready, start, or finish. Among the possible traces, a wide variety of traces may achieve similar performance, and identifying the parts in a trace that affect performance is difficult. This complicates understanding the superiority of one schedule over another. Even with an efficient method to pinpoint problems in schedules, the mapping between problematic scheduling decisions and particularities of a scheduling heuristic would ideally be automated.

6.2.4. Trust and Practicality

Finally, human factors play a crucial role concerning the practicality of automatic resource allocation. While users will undoubtedly be happy to be unburdened from the challenge of resource usage estimation, learned estimates also introduce new challenges:

- Users have to get used to monitoring a prediction model's performance rather than the accuracy of their estimates. In case model performance is deficient, fixing a prediction system is much harder than fixing a user estimate.
- Users and system developers have to be convinced that a prediction model is reliable. The foundations for trust are prediction accuracy, robustness, and simulation studies.
- A prediction model's decisions may seem counter-intuitive to users. Ideally, predictions would be explainable to support their acceptance.

Guarantees to outperform user estimates would help facilitate the adoption of a prediction model. However, providing guarantees is not possible without making assumptions, e. g., on the predictability of the workflows or the quality of the user estimates. Artificial or violated assumptions may render guarantees infeasible in practice. However, striving for robustness is essential. This includes mechanisms to anticipate and avoid model failure, e. g., preventing extreme predictions by falling back to user estimates. In the long run, the net gains provided by predictive resource allocation methods will be the crucial factor for their adoption. This thesis suggests that the potential of automated resource allocation is sufficient.

Appendices

A. Workflow Management Systems

Section 2.1.3 provides a brief overview of commonly used workflow management systems. The assessment of common use is based on which workflow management systems have been covered by eleven sources published between 2015 and 2019. As an additional source, the list [Amstutz et al., 2019] of workflow management systems implementing the Common Workflow Language [Amstutz et al., 2016] has been used. At the time of writing, it appears to be the most promising standardization effort for scientific workflow languages and scientific workflow management systems that implement it are expected to exhibit a certain level of maturity.

This appendix is structured as follows. Section A.1 provides an overview of the criteria used to compare and classify scientific workflow management systems in the eleven sources. Section A.2 provides the full list of scientific workflow management systems covered by the eleven papers and the Common Workflow Language implementers list. Section A.3 provides the most extensive list of currently available software covering 211 workflow management systems, tools, and languages.

A.1. Workflow Management System Traits

Source	Criterion	Possible values
[Ferreira da Silva et al., 2017]	Workflow Execution Model	Sequential, Concurrent, Iterative, Tightly coupled, External steering
	Heterogeneous Computing Environments	Co-location, External location, In situ
	Data Access Methods	Memory, Messages, Local disk, Shared file system, Object store, Other remote storage
[Leipzig, 2017]	Syntax	Explicit, Implicit
	Paradigm	Class, Convention, Configuration
	Interaction	Cli, Server, Workbench, Commercial, Cloud, Cloud API
	Ease of Development	*Rating*
	Ease of Use	*Rating*
	Performance	*Rating*
[Atkinson et al., 2017]	Advances or major achievements	*Text*

Table A.1 continued from previous page

Source	Criterion	Possible values
[Liew et al., 2017]	Processing element	Executable program, Web service
	Optimization stage	Build time, Run time
	User interface	Textual, Graphical
	Data processing model	Stream, Bulk
[Khan et al., 2017]	Workflow structure	Dag, Dcg
	Static scheduling	*Boolean*
	Information sharing	*Boolean*
	User interface	Textual, Gui, Desktop, Web
	Special feature	Textual
[Liu et al., 2015]	Workflow Structure	Dag, Dcg
	Workflow sharing	*Boolean*
	User interface	Textual, Graphical
	Parallelism	Data, Activity, Independent, Hybrid
	Scheduling	Static, Dynamic, Hybrid
[Di Tommaso et al., 2017]	Platform	JVM, Python
	Native task support	*Boolean*
	Common workflow language	*Boolean*
	Stream processing	*Boolean*
	Dynamic branch evaluation	*Boolean*
	Code sharing integration	*Boolean*
	Workflow modules	*Boolean*
	Workflow versioning	*Boolean*
	Automatic error failover	*Boolean*
	Graphical user interface	*Boolean*
	DAG rendering	*Boolean*
	Container management	Docker, Singularity
	Multi-scale container	*Boolean*
	Built-in batch schedulers	Univa Grid Engine, PBS/Torque, LSF, SLURM, HTCondor
	Built-in distributed cluster	Apache Ignite, Apache Spark, Kubernetes, Apache Mesos
	Built-in AWS Support	*Boolean*
[Wang and Peng, 2019]	Language	Python based, Groovy flavoured, Gnu make style
	User interface	Cli, Jupyter notebook, Gui
	File format	Json, Xml, Source code, Jupyter notebook, Yaml

Table A.1 continued from previous page

Source	Criterion	Possible values
	Integrated development environment	*Boolean*
	Dag construction	Explicit, Implicit
	Stream processing	*Boolean*
	Subworkflows	*Boolean*
	Atomic write	*Boolean*
	Named input/output	*Boolean*
	Modify and resume	*Boolean*
	Built-in remote execution	*Boolean*
	Task monitoring	Command line, Gui, Notebook, Reports, Traces, Event notification
	Workflow Orientation	Process, Output
	Built-in container support	Docker, Singularity
	Built-in batch schedulers	PBS/Torque, LSF, SLURM, HTCondor,
	Cloud storage	*Boolean*
[Bux, 2017]	Parallelism	Task, Data
	Distribution	Batch scheduler, Standalone
	Scheduling	Static, Knowledge-free
[Boulakia et al., 2017]	Workflow language	Xml, Scufl 2, Json, Python source, Groovy source
	Interoperability support	Prov, Cwl
	Local lib tools command	Java, R, Python, Groovy
	Tools	Command line, Remote service
	Access to source code of workflow steps	Yes, Local tools only
	Workflow annotation	Steps, Workflow
	Language	Taverna-prov, Json, Provone, Prov, Opm, Source code
	Standards used	Prov, Opm
	Nested workflows	*Boolean*
	System-wide packaging	Docker, Conda, Vagrant, Reprozip
	Scientific-wide packaging	Research objects, Isa
[Mork et al., 2015]	Domain publications	*Integer value*
	CompSci publications	*Integer value*
	Tools and applications	*Integer value*
	Primary publications from tools	*Integer value*

Table A.1.: The workflow management system comparison criteria used in the comparison studies listed in Table 2.1.

A.2. Common Workflow Management Systems

The following table lists common scientific workflow management systems, where the term common refers to citation in survey papers comparing workflow management systems. The table is based on the 12 sources mentioned in the introduction to this appendix.

System	Compared to other workflow management systems in
Galaxy	[Atkinson et al., 2017], [Boulakia et al., 2017], [Bux, 2017], CWL Implementers List, [Di Tommaso et al., 2017], [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Leipzig, 2017], [Liu et al., 2015], [Mork et al., 2015], [Wang and Peng, 2019]
Taverna	[Atkinson et al., 2017], [Boulakia et al., 2017], [Bux, 2017], CWL Implementers List, [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Leipzig, 2017], [Liew et al., 2017], [Liu et al., 2015], [Mork et al., 2015]
Pegasus	[Atkinson et al., 2017], [Bux, 2017], [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Leipzig, 2017], [Liew et al., 2017], [Liu et al., 2015], [Mork et al., 2015]
Kepler	[Atkinson et al., 2017], [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Liew et al., 2017], [Liu et al., 2015], [Mork et al., 2015]
Swift	[Atkinson et al., 2017], [Bux, 2017], [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Liew et al., 2017], [Liu et al., 2015]
Nextflow	[Boulakia et al., 2017], [Di Tommaso et al., 2017], [Ferreira da Silva et al., 2017], [Leipzig, 2017], [Wang and Peng, 2019]
Triana	[Atkinson et al., 2017], [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Liu et al., 2015], [Mork et al., 2015]
Askalon	[Atkinson et al., 2017], [Ferreira da Silva et al., 2017], [Khan et al., 2017], [Liu et al., 2015]
Snakemake	[Bux, 2017], [Di Tommaso et al., 2017], [Leipzig, 2017], [Wang and Peng, 2019]
Airavata	[Atkinson et al., 2017], [Ferreira da Silva et al., 2017], [Liew et al., 2017]
Bpipe	[Di Tommaso et al., 2017], [Leipzig, 2017], [Wang and Peng, 2019]
Knime	[Atkinson et al., 2017], [Bux, 2017], [Liew et al., 2017]
Toil	CWL Implementers List, [Di Tommaso et al., 2017], [Leipzig, 2017]
Arvados	CWL Implementers List, [Leipzig, 2017]
Chiron	[Khan et al., 2017], [Liu et al., 2015]
Dispel4py	[Atkinson et al., 2017], [Ferreira da Silva et al., 2017]
Moteur	[Atkinson et al., 2017], [Ferreira da Silva et al., 2017]
Vistrails	[Boulakia et al., 2017], [Mork et al., 2015]
Wpg	[Khan et al., 2017], [Liu et al., 2015]
Adios	[Ferreira da Silva et al., 2017]
Agave	[Leipzig, 2017]
Airflow	CWL Implementers List
Awe	CWL Implementers List

Table A.2 continued from previous page

System	Compared to other workflow management systems in
BigDataScript	[Leipzig, 2017]
Bobolang	[Ferreira da Silva et al., 2017]
Calrissian	CWL Implementers List
Consonance	CWL Implementers List
Cuneiform	[Bux, 2017]
Cwl-Tes	CWL Implementers List
Cwlexec	CWL Implementers List
Dnanexus	[Leipzig, 2017]
Fireworks	[Ferreira da Silva et al., 2017]
Guse	[Atkinson et al., 2017]
Hi-WAY	[Bux, 2017]
Luigi	[Leipzig, 2017]
Makeflow	[Ferreira da Silva et al., 2017]
Meandre	[Liew et al., 2017]
Openalea	[Boulakia et al., 2017]
Queue	[Leipzig, 2017]
Rapidminer	[Mork et al., 2015]
Reana	CWL Implementers List
Ruffus	[Leipzig, 2017]
Sevenbridges	[Leipzig, 2017]
SoS	[Wang and Peng, 2019]
Wings	[Atkinson et al., 2017]
Xenon	CWL Implementers List
Yacle	CWL Implementers List

Table A.2.: All workflow management systems appearing in one of the scientific workflow management comparison studies listed in Table 2.1.

A.3. Workflow Management Systems, Languages, and Tools

The following is a snapshot of a publicly maintained list[1] of "computational data analysis workflow systems". However, the list provides no definition of what a computational data analysis workflow system is. In fact, some of the entries may refer to workflow languages (such as YAWL), software packages, or even deprecated products (Yahoo! Pipes). It is however, one of the most comprehensive sources of workflow management system related software and is, at the time of writing, actively maintained by the community.

Adage
AdaptiveMd
Agave
Agua
Airavata
Airflow
Anduril
Anvaya
Apache Beam
Apache NiFi
Archivematica
Argo
Arvados
AWE
Azkaban
BASTet
Bcbio
BigDataScript
BioCloud
BioDSL
BioDT
BioMAJ
BioMake
BioMoby
Bionode Watermill
Biopet
BioPig
Biopipe
Bioshake
BIOVIA Pipeline Pilot Overview
BioWBI
BioWMS
BioWorkbench
BPipe
CGAT-core
Chipster
Chronos
Civet
CLC Genomics Workbench
Clinical Trial Processor
CLOSHA
CloudDOE
Cloudgene
CloudSlang
Cluster Flow
COMBUSTI/O
Consonance
Conveyor
Copernicus
Cosmos
Cpipe
Cromwell
Cuisine Framework
Cumulus
Cuneiform
Cylc
Cyrille2
Dagman
Dagobah
DALiuGE
Dask
Dgsh
Digdag
DiscoveryEnvironment
dispel4py
DNANexus
Dog
Dray
ECFLOW
EDGE bioinformatics
EGene
End of Day
Ensembl Hive
ENVI Task Engine
Eoulsan
Ergatis
FASTR
Fireworks.
Flowr
Flowstone
Fractalide
Galaxy
GATE Cloud
GATK Queue
GC3Pie
GenePattern
GenomeVIP
GenPipes
Ginflow
GNU Guix Workflow Language
GridSAM
Grid WorkFlow
HI-WAY
HyperLoom
Idseq-dag
IMP
iRODS Rule Language
JMS
Jobber
JobCenter
JTracker
JX Workflow
Kapacitor
Kepler
Ketrew
KLIKO
Knime
Kronos
LabView
Leaf
LONI Pipeline
Loom
LSST Data Management
Luigi
Makeflow
Martian
Max/MSP
Meshroom
Metapipe
Microbase
Mistral
mpipe

[1] https://s.apache.org/existing-workflow-systems

MyOpenLab
NeatSeq-Flow
Nephele
Nesoni
NextFlow
NGLess
Niassa
nipype
NoFlo
Noodles
OCCAM
omictools
Oozie
OpenAlea
OpenMOLE
Ophidia
Overseer
Pachyderm
PaPy
Parsl
phpflo
Pinball
PipelineDog
Piper
Plumbing and Graph
Porcupine
Prefect
Produce
Pwrake
Pyflow
pypeFLOW
Pypeline
pypipegraph
QosCosGrid
qsubsec
QuickNGS
Reana
Reflow
remake
Resolwe
Roddy
Ruffus
S4M
SBpipe
SciApps
SciFlo
SciFlow
SciFloware
SciLuigi
SCIPION
SciPipe
SeqPig
SHIWA
SIBIOS
Skam
Snakemake
Squonk
Stacks
Stimela
Stoa
Sushi
Swift
Tavaxy
Taverna
TIGR
Tiny Cloud Engine
TOGGLe
Toil
TOPPAS
TREVA
Triana
Triquetrum
UNICORE
Unipro UGENE
VIBE
VisTrails
Watchdog
WDL
WebLicht
WEP
Wildfire
Wings
WopMars
Xbowflow
XiP
Yabi
Yahoo! Pipes
YAWL
YesWorkflow

B. Supplemental Figures

B.1. Supplemental Figures for Chapter 3

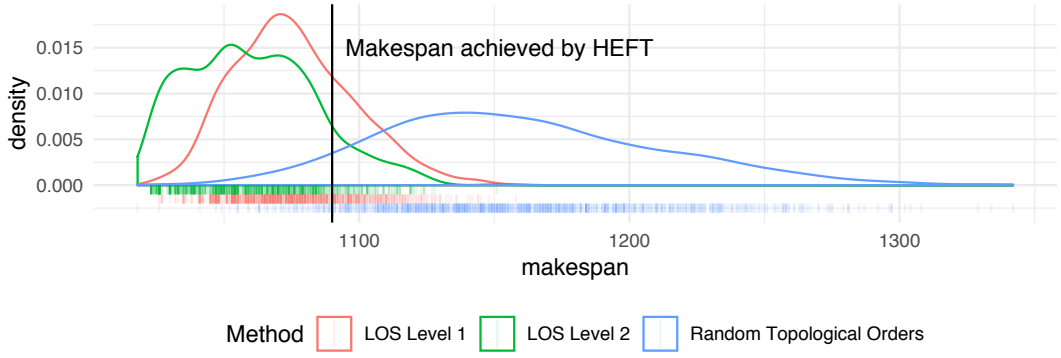

Figure B.1.: Distributions of makespans of L-Orders sampled from single levels of a reference L-Order. Completely random topological orders have a lower probability of outperforming HEFT, which demonstrates the benefit of focusing variations on levels of a dag. The underlying dag (n=50, p=3) was generated with the method described in Section 3.2.1.

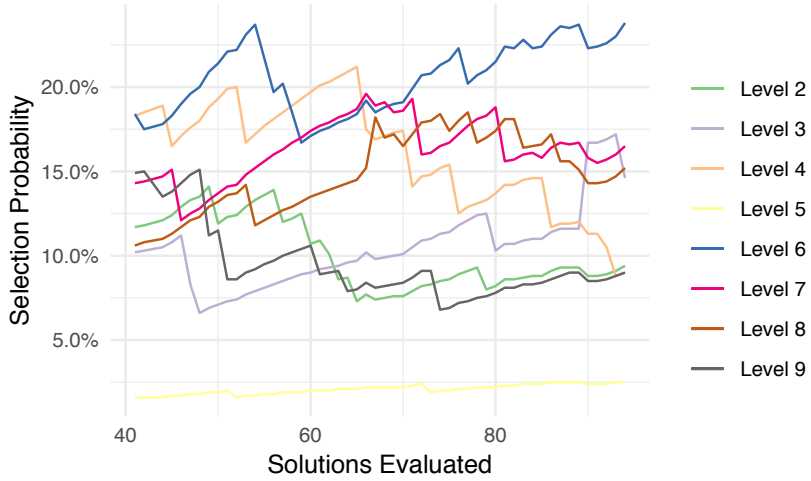

Figure B.2.: Exemplary evolution of level selection probabilities during a single exploitation phase. Note that although level 6 has the highest selection probability most of the time, the randomized level selection scheme makes sure that other levels are also sampled.

B.2. Supplemental Figures for Chapter 4

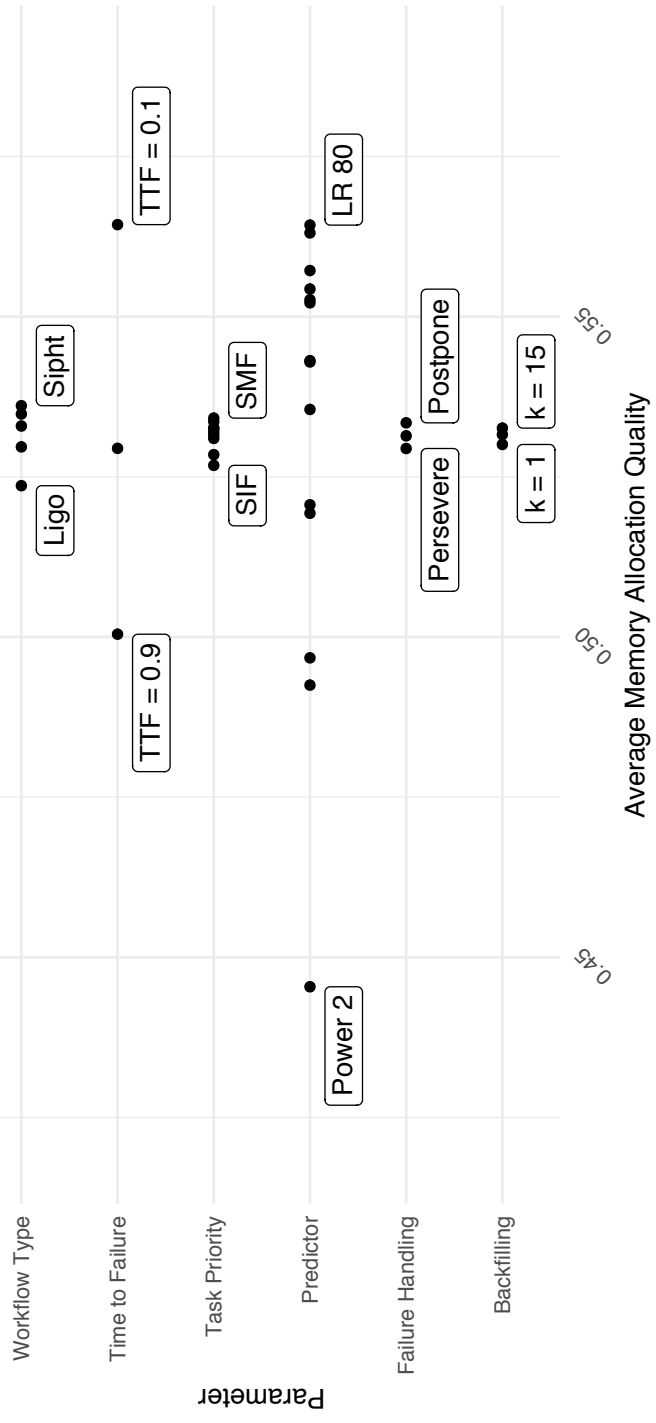

Figure B.3.: Parameter impact on memory allocation quality, as measured by averaging the memory allocation quality of all simulations using a specific value for a specific parameter. The best and worst values for each parameter are labelled.

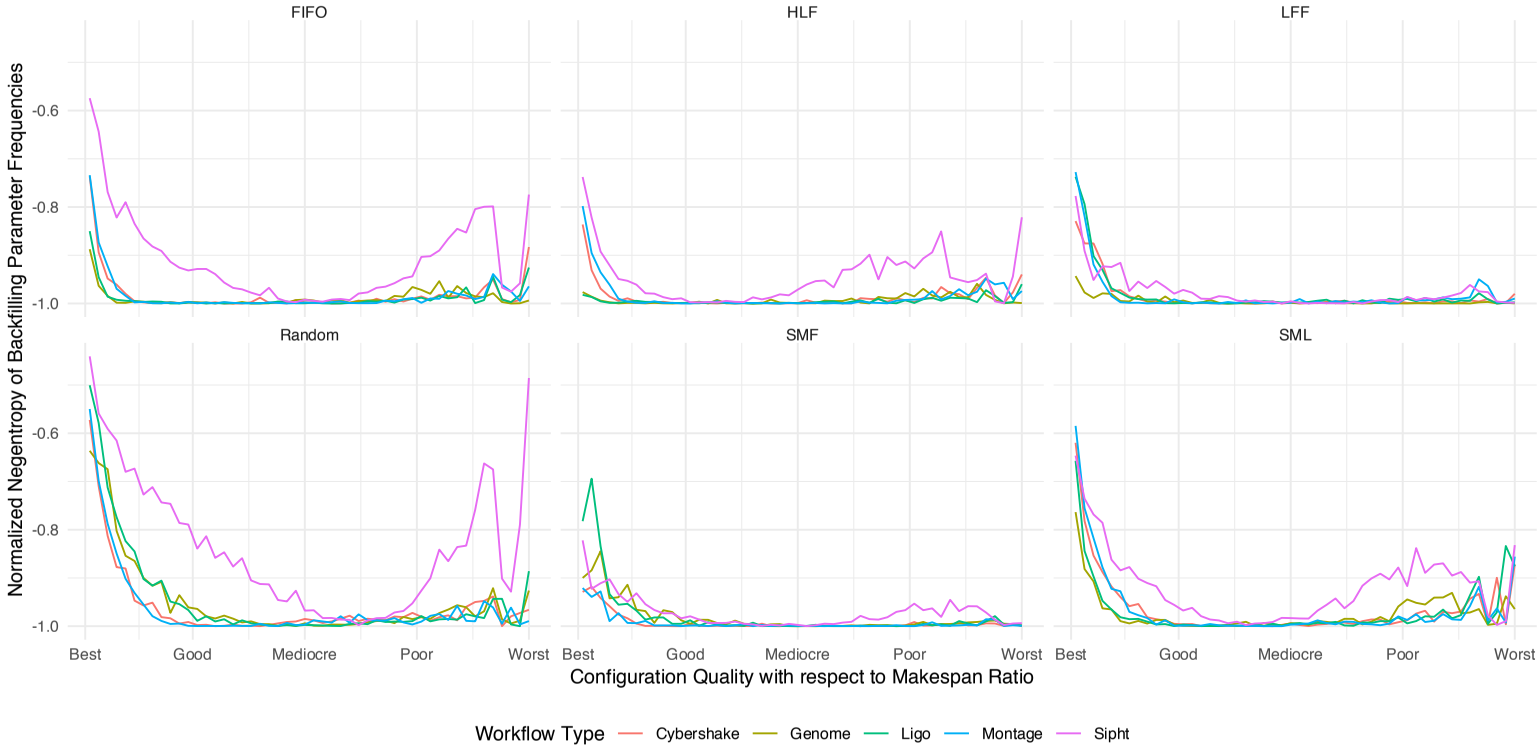

Figure B.4.: Backfilling parameter dominance as a function of performance. The negative entropy measures the degree to which a single parameter prevails in a set of results. The figure shows the results partitioned into the best 1%, second-best 1%, etc. The Random heuristic favors certain choices of the parameter, as indicated by high negentropy values. In contrast, LFF is mostly insensitive to backfilling; especially in the mediocre and poor performing simulations, all parameter choices are equally frequent (negentropy close to zero). Note how this also depends on workflow topology; Sipht often deviates from the other topologies.

B.3. Supplemental Figures for Chapter 5

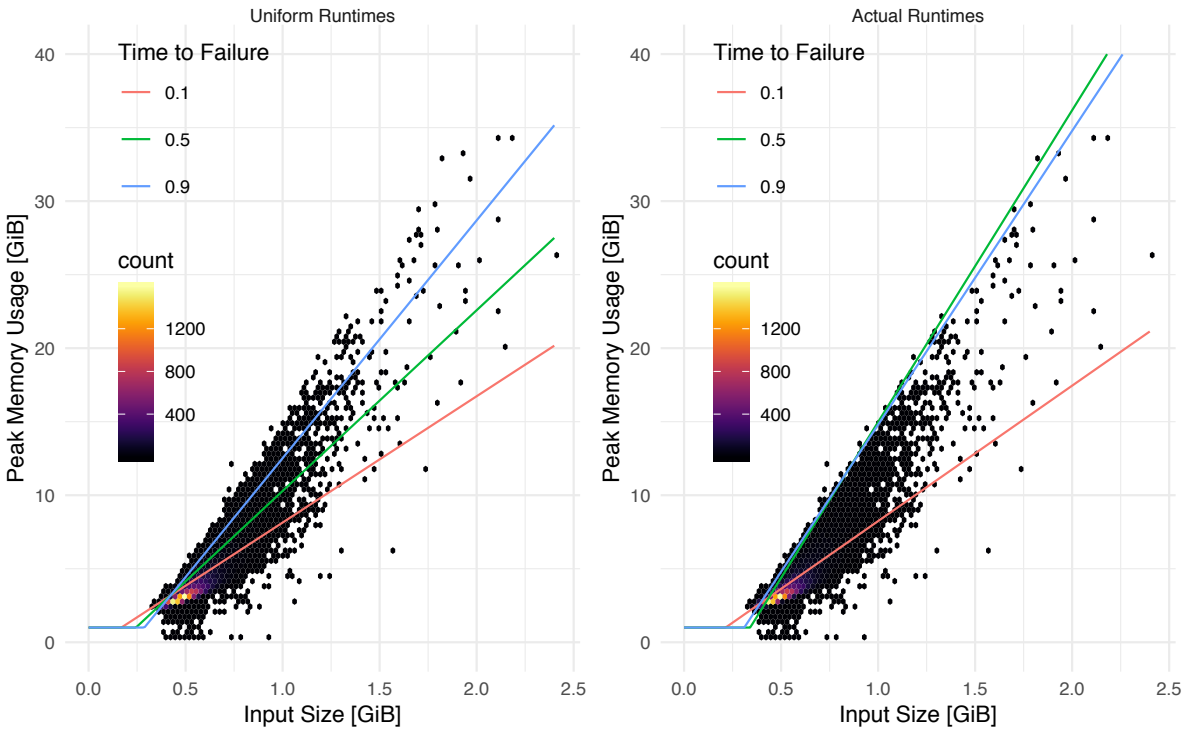

Figure B.5.: Optimal model parameters depend on the run times of the tasks as well as the time to failure parameter. On the left side, all tasks are assumed to have the same run time. On the right side, model parameters are optimized for task run times as recorded in the IceCube data.

Figure B.5 illustrates the impact of the run times and time to failure on model parameters. The plot shows the input sizes and peak memory usages of approximately 25 000 tasks from the IceCube data set. The red, green, and blue lines indicate optimized model parameters for different circumstances. MAQ varies between 68% and 74% for all models, but the chosen slopes and intercepts differ strongly.

In the case of uniform task run times, it is better to let large tasks fail on their first attempt to reduce the oversizing wastage for smaller tasks. In the IceCube data set, tasks that require more memory also tend to run longer, increasing the undersizing wastage for the large-memory tasks. On the real-world data, memory allocation quality is maximized by more conservative models, as shown in the right part of the figure.

C. Random Linear Models

In Section 4.2.1, a method to generate random data points that follow a linear model is proposed. It is claimed that splitting the desired variance μ_y according to a parameter l results in an expected Pearson correlation coefficient of $\sqrt{l}$. Recall that the parameters of the normally distributed independent variable X are chosen (according to Equations 4.3 and following) such that the dependent variable Y has desired mean μ_y and desired variance σ_y^2 while being centered around a regression line with slope θ_1 and intercept θ_0.

$$X \sim \mathcal{N}\left(\underbrace{\frac{\mu_y - \theta_0}{\theta_1}}_{\mu_x}, \underbrace{l\frac{\sigma_y^2}{\theta_1^2}}_{\sigma_x^2}\right) \tag{C.1}$$

$$Y \sim \theta_1 X + \theta_0 + \mathcal{N}(0, \underbrace{(1-l)\sigma_y^2}_{\sigma_e^2}) \tag{C.2}$$

The following proof shows that

$$\rho_{X,Y} = \frac{\mathrm{Cov}(X,Y)}{\sigma_x \sigma_y} = \sqrt{l}$$

The derivation starts from $\mathrm{Cov}(X,Y) = \mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y]$. The first term is:

$$\begin{aligned}\mathrm{E}[XY] &= \mathrm{E}[\theta_1 X^2 + \theta_0 X + \mathcal{N}(0,\sigma_e^2)X] \\ &= \theta_1\,\mathrm{E}[X^2] + \theta_0\mu_x + \mathrm{E}[\mathcal{N}(0,\sigma_e^2)X]\end{aligned}$$

The expectation $\mathrm{E}[\mathcal{N}(\mu,\sigma^2)^2]$ of a squared normal can be derived using the χ^2 distribution:

$$\begin{aligned}\mathrm{E}[\mathcal{N}(\mu,\sigma^2)^2] &= \mathrm{E}[(\sigma\mathcal{N}(0,1) + \mu)^2] \\ &= \mathrm{E}[\sigma^2\mathcal{N}(0,1)^2 + 2\sigma\mu\mathcal{N}(0,1) + \mu^2] \\ &= \sigma^2 \underbrace{\mathrm{E}[\mathcal{N}(0,1)^2]}_{=\mathrm{E}[\chi_1^2]=1} + 0 + \mu^2 = \sigma^2 + \mu^2\end{aligned}$$

Applying this to $\mathrm{E}[X^2]$ and substituting σ_x^2 according to Equation C.1:

$$\mathrm{E}[XY] = \theta_1\left(l\frac{\sigma_y^2}{\theta_1^2} + \mu_x^2\right) + \theta_0\mu_x + \mathrm{E}[\mathcal{N}(0,\sigma_e^2)X]$$

Since the errors and the inputs are independent, $\mathrm{E}[\mathcal{N}(0, \sigma_e^2)X] = \mathrm{E}[\mathcal{N}(0, \sigma_e^2)]\,\mathrm{E}[X] = 0$. Thus, the covariance of the two variables can be written as:

$$\begin{aligned}
\mathrm{Cov}(X, Y) &= \mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y] \\
&= \theta_1(l\frac{\sigma_y^2}{\theta_1^2} + \mu_x^2) + \theta_0\mu_x - \mu_x\mu_y \\
&= \theta_1 l\frac{\sigma_y^2}{\theta_1^2} + \mu_x(\underbrace{\theta_1\mu_x}_{=\mu_y-\theta_0} + \theta_0 - \mu_y) \\
&= \theta_1 l\frac{\sigma_y^2}{\theta_1^2} = l\sigma_y^2/\theta_1
\end{aligned}$$

Dividing by the product $\sigma_x\sigma_y$ of the standard deviations gives:

$$\begin{aligned}
\rho_{X,Y} &= \frac{l\sigma_y^2/\theta_1}{\sigma_x\sigma_y} \\
&= \frac{l\sigma_y^2/\theta_1}{\sqrt{l\sigma_y^2/\theta_1^2}\sigma_y} \\
&= \frac{l\sigma_y^2}{\sqrt{l}\sigma_y^2} \\
&= \sqrt{l} \quad \square
\end{aligned}$$

D. Workflow Partitioning

Chapter 4 uses lower bounds to relate absolute workflow execution durations to minimum execution times. A workflow partitioning method is applied to combine two lower bounds into a tighter one.

The goal is to partition the tasks of a workflow into a sequence $S_1, ..., S_n \subseteq V$ of task sets such that every task in set S_i depends on every task in S_{i-1}, i. e., no tasks in S_i can be executed in parallel to a task in S_{i-1}. Since every task in S_{i-1} has to finish before any task in S_i can be started, a lower bound on the makespan of the induced subgraph of S_{i-1} is a lower bound on the starting time of the induced subgraph of S_i. The workflow can then be treated as a sequence of sub-workflows, as shown in Figure D.1.

Example

The advantage of partitioning a workflow prior to computing lower bounds is that the execution times of different partitions can be bound by different criteria. Consider the workflow in Figure D.1, which can be partitioned in two partitions. Assume $C = 2$ processors and no memory usage for the sake of simplicity. For the first partition, the total work lower bound $\mathrm{TW}(S_1) = \sum \tau / C = 8/2 = 4$ is tighter than the critical path lower bound $\mathrm{CP}(S_1) = 2$. For the second partition, the critical path lower bound $\mathrm{CP}(S_2) = 10$ is tighter than the total work lower bound $\mathrm{TW}(S_2) = 10/2 = 5$. By partitioning, the best lower bound can be used for different parts of the graph, giving a total lower bound of $\mathrm{PB}(G) = \mathrm{TW}(S_1) + \mathrm{CP}(S_2) = 4 + 10 = 14$. Without partitioning on the other hand, the overall lower bounds are not as tight: $\mathrm{TW}(G) = 18/2 = 9$ and $\mathrm{CP}(G) = 12$.

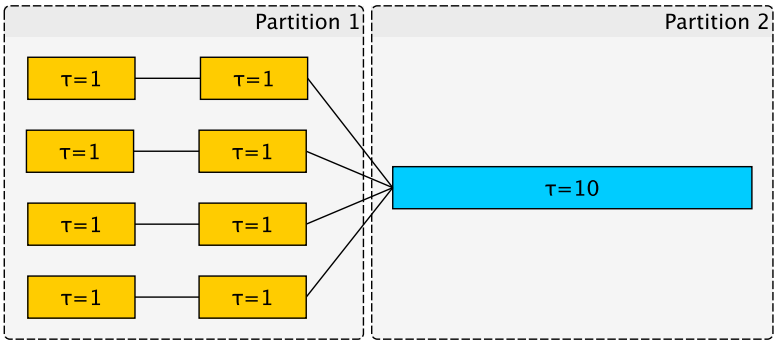

Figure D.1.: Example partitioning of a workflow with 9 tasks and specified run times τ. Edges are directed from left to right.

Algorithm

A sequence $S_1, ..., S_n \subseteq V$ of maximal size n can be found in time $\mathcal{O}(|V| + |E|)$. Let $G = (V, E)$ denote a directed acyclic graph. Let L_i denote all tasks in the graph with level i.

$$L_i = \{T \in V \mid \text{level}(T) = i\} \tag{D.1}$$

Let l denote the maximum level of a task in the graph. Since levels are non-negative,

$$\bigcup_{i=0}^{l} L_i = V \tag{D.2}$$

The idea is to construct the partitions by processing the tasks level-wise. The central observation is that the levels of the tasks in a partition S_{i-1} are all larger than the levels of the tasks in the next partition S_i. To see this, recall that every task $T_j \in S_i$ depends on every task $T_i \in S_{i-1}$, i. e., $T_i \prec T_j$ and the definition of the level of a task

$$\text{level}(T_i) = \begin{cases} 0 & T_i \text{ is an exit task} \\ 1 + \max\{\text{level}(T_j) \mid T_i \prec T_j\} & \text{otherwise} \end{cases}$$

Thus, by iterating over the tasks level-wise and checking if all tasks of the current level depend on all tasks of the previous level suffices to partition the graph. If there is a task T^* in the current level L_i that does not depend on every task in L_{i+1}, the current partition can be expanded by including all tasks in level L_i. The reasoning is that T^* is part of the current partition (because it can be executed in parallel with a task in the current partition) and thus every task that can be executed in parallel with T^* is also part of the current partition. Tasks with the same level cannot depend on each other: otherwise, one task could increase its level by adding one or more to the level of the other. Thus all tasks in L_i can be executed in parallel and can thus be added to the current partition.

It is assumed that the graph does not allow multiple edges between vertices. Then, to check whether a task in L_i depends on all tasks with level $i + 1$, it suffices to count the number of predecessors with level $i + 1$ and compare it to the number of tasks with level $i + 1$. This leads to an overall complexity of $\mathcal{O}(|V| + |E|)$ since every task $T \in V$ is visited exactly once when examining its level. For deciding whether it belongs to a new partition or not, it suffices to evaluate the level of each of its predecessors, such that every edge in the graph is visited exactly once. The levels can also be computed in $\mathcal{O}(|V| + |E|)$ time.

```
Method Partition(Directed Acyclic Graph G):
    // S is the set of partitions
    S ← ∅;
    // S_k is the partition currently constructed
    S_k ← L_l;
    for i = l − 1 down to 0 do
        newPartition ← true;
        // The task has to depend on every task in the previous level,
           or parallelism is possible.
        for Task t ∈ L_i do
            // Check that the number of predecessors with level i + 1
               equals the size of level i + 1
            if |{p ∈ pred(t) | level(p) = i + 1}| < |L_{i+1}| then
                newPartition ← false;
                break;
            end
        end
        if newPartition then
            // The current partition S_k cannot be further expanded
            S ← S ∪ S_k;
            // All tasks in L_i depend on all tasks in S_k and start the
               next partition
            S_k ← L_i;
        end
        else
            // All tasks in the current level belong to the current
               partition S_k
            S_k ← S_k ∪ L_i;
        end
    end
    // The partition including the task with level 0 cannot be further
       expanded
    S ← S ∪ S_k;
    return S
end
```

Algorithm 5: The partition algorithm returns a set of task sets such that the sum of the lower bounds on execution time on the induced sub-workflows of the task sets is a lower bound on the execution time of the overall workflow.

Bibliography

[Adaptive Computing Enterprises, Inc., 2019a] Adaptive Computing Enterprises, Inc. (2019a). Moab HPC Suite http://www.adaptivecomputing.com/moab-hpc-basic-edition/.

[Adaptive Computing Enterprises, Inc., 2019b] Adaptive Computing Enterprises, Inc. (2019b). TORQUE Resource Manager http://www.adaptivecomputing.com/products/torque/.

[Agullo et al., 2016] Agullo, E., Beaumont, O., Eyraud-Dubois, L., and Kumar, S. (2016). Are Static Schedules so Bad? A Case Study on Cholesky Factorization. In *Proceedings of the 30th International Parallel and Distributed Processing Symposium*, pages 1021–1030. IEEE.

[Albrecht et al., 2012] Albrecht, M., Donnelly, P., Bui, P., and Thain, D. (2012). Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *the 1st ACM SIGMOD Workshop*, pages 1–13, New York, New York, USA. ACM Press.

[Alipourfard et al., 2017] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. (2017). CherryPick - Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[Almeida et al., 2019] Almeida, A., Mitchell, A. L., Boland, M., Forster, S. C., Gloor, G. B., Tarkowska, A., Lawley, T. D., and Finn, R. D. (2019). A new genomic blueprint of the human gut microbiota. *Nature*, 568(7753):499–504.

[Altair Engineering, Inc., 2019] Altair Engineering, Inc. (2019). PBS Professional Open Source Project https://www.pbspro.org/.

[Amstutz et al., 2019] Amstutz, P., Chilton, J., Crusoe, M. R., Dusenbery, B. D., Gentry, J., Ménager, H., and Soiland-Reyes, S. (2019). Common Workflow Language https://www.commonwl.org/.

[Amstutz et al., 2016] Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Ménager, H., Nedeljkovich, M., Scales, M., Soiland-Reyes, S., and Stojanovic, L. (2016). Common Workflow Language, v1.0 https://escholarship.org/uc/item/25z538jj.

[Andresen et al., 2018] Andresen, D., Hsu, W., Yang, H., and Okanlawon, A. (2018). Machine Learning for Predictive Analytics of Compute Cluster Jobs. In *arXiv:1806.01116*.

[Arabnejad and Barbosa, 2014] Arabnejad, H. and Barbosa, J. G. (2014). List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Trans. Parallel Distrib. Syst.*, 25(3):682–694.

[Armstrong et al., 1998] Armstrong, R., Hensgen, D., and Kidd, T. (1998). The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proceedings of the 7th Heterogeneous Computing Workshop*, pages 79–87.

[Atef et al., 2017] Atef, A., Hagras, T., Mahdy, Y. B., and Janeček, J. (2017). Lower-bound complexity algorithm for task scheduling on heterogeneous grid. *Computing*, 99(11):1125–1145.

[Atkeson et al., 1997] Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73.

[Atkinson et al., 2017] Atkinson, M., Gesing, S., Montagnat, J., and Taylor, I. J. (2017). Scientific workflows: Past, present and future. *Future Generation Computing Systems*, 75:216–227.

[Bittencourt et al., 2010] Bittencourt, L. F., Sakellariou, R., and Madeira, E. R. M. (2010). DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society.

[Bittencourt et al., 2012] Bittencourt, L. F., Sakellariou, R., and Madeira, E. R. M. (2012). Using relative costs in workflow scheduling to cope with input data uncertainty. In *Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science*, pages 1–6. ACM Press.

[Blankenberg et al., 2010] Blankenberg, D., Von Kuster, G., Coraor, N., Ananda, G., Lazarus, R., Mangan, M., Nekrutenko, A., and Taylor, J. (2010). Galaxy: A Web-Based Genome Analysis Tool for Experimentalists. *Current Protocols in Molecular Biology*, 89(1).

[Boulakia et al., 2017] Boulakia, S. C., Belhajjame, K., Collin, O., Chopard, J., Froidevaux, C., Gaignard, A., Hinsen, K., Larmande, P., Le Bras, Y., Lemoine, F., Mareuil, F., Ménager, H., Pradal, C., and Blanchet, C. (2017). Scientific workflows for computational reproducibility in the life sciences - Status, challenges and opportunities. *Future Generation Computing Systems*, 75:284–298.

[Box et al., 2015] Box, G. E. P., Jenkins, G. M., Reinsel, G. C., and Ljung, G. M. (2015). *Time series analysis. Forecasting and control*. Holden-Day Series in Time Series Analysis, 4th edition.

[Brandt et al., 2015] Brandt, J., Bux, M., and Leser, U. (2015). Cuneiform: A Functional Language for Large Scale Scientific Data Analysis. *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference*, pages 7–16.

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and regression trees*. Wadsworth Statistics/Probability Series. Wadsworth Advanced Books and Software, Belmont, CA.

[Budrikis, 2019] Budrikis, Z. (2019). Nuts and bolts of seeing a black hole. *Nature Reviews Physics*, 1(5):305–305.

[Bux, 2017] Bux, M. (2017). *Scientific Workflows for Hadoop*. PhD thesis, Humboldt-Universität zu Berlin.

[Bux et al., 2017] Bux, M., Brandt, J. r., Witt, C., Dowling, J., and Leser, U. (2017). Hi-WAY: Execution of scientific workflows on hadoop YARN. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, pages 668–679.

[Bux and Leser, 2015] Bux, M. and Leser, U. (2015). DynamicCloudSim: Simulating heterogeneity in computational clouds. *Future Generation Computing Systems*, 46:85–99.

[Cai et al., 2018] Cai, H., Zheng, V. W., and Chang, K. C.-C. (2018). A Comprehensive Survey of Graph Embedding - Problems, Techniques, and Applications. *IEEE Trans. Knowl. Data Eng.*, 30(9):1616–1637.

[Calheiros et al., 2010] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2010). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50.

[Casanova et al., 2008] Casanova, H., Legrand, A., and Robert, Y. (2008). *Parallel Algorithms*. CRC Press.

[Casanova et al., 2000] Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. (2000). Heuristics for scheduling parameter sweep applications in grid environments. *9th Heterogeneous Computing Workshop (HCW 2000)*, pages 349–363.

[Casanova et al., 2019] Casanova, H., Pandey, S., Oeth, J., Tanaka, R., Suter, F., and Ferreira da Silva, R. (2019). WRENCH: A Framework for Simulating Workflow Management Systems. In *Proceedings of the 13th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 74–85. IEEE.

[Chakravarthi and Vijayakumar, 2018] Chakravarthi, K. K. and Vijayakumar, V. (2018). Workflow scheduling techniques and algorithms in IaaS cloud: A survey. *International Journal of Electrical and Computer Engineering*, 81(2):1256–1268.

[Chang, 2015] Chang, J. (2015). Core services: Reward bioinformaticians. *Nature*, 520(7546):151–152.

[Chatzopoulos et al., 2016] Chatzopoulos, G., Dragojevic, A., and Guerraoui, R. (2016). ESTIMA: Extrapolating Scalability of In-Memory Applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–11. ACM Press.

[Chen and Deelman, 2012] Chen, W. and Deelman, E. (2012). WorkflowSim: A toolkit for simulating scientific workflows in distributed environments. In *Proceedings of the 8th International Conference on E-Science*, pages 1–8. IEEE.

[Chen et al., 2017] Chen, W., Xu, Y., and Wu, X. (2017). Deep Reinforcement Learning for Multi-Resource Multi-Machine Job Scheduling. arXiv:1711.07440 [cs.DC].

[Cordeiro et al., 2010] Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J. M., and Wagner, F. (2010). Random graph generation for scheduling simulations. In *SIMUTools 2010 - 3rd International ICST Conference on Simulation Tools and Techniques*. Universite Grenoble Alpes, Grenoble, France, ICST.

[Cristianini and Shawe-Taylor, 2009] Cristianini, N. and Shawe-Taylor, J. (2009). *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, Cambridge.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[Deelman et al., 2015] Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., and Wenger, K. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.

[Delimitrou and Kozyrakis, 2014] Delimitrou, C. and Kozyrakis, C. (2014). Quasar: Resource-efficient and QoS-aware cluster management. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–143. ACM Press.

[Di Tommaso et al., 2017] Di Tommaso, P., Chatzou, M., Floden, E. W., Barja, P. P., Palumbo, E., and Notredame, C. (2017). Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319.

[Drozdowski, 2010] Drozdowski, M. (2010). *Scheduling for Parallel Processing*. Springer Science & Business Media.

[Duplyakin et al., 2018] Duplyakin, D., Brown, J., and Calhoun, D. (2018). Evaluating active learning with cost and memory awareness. In *Proceedings of the 32nd International Parallel and Distributed Processing Symposium*, pages 214–223. IEEE.

[Ferreira da Silva et al., 2014] Ferreira da Silva, R., Chen, W., Juve, G., and Vahi, K. (2014). Community resources for enabling research in distributed scientific workflows. In *Proceedings of the International Conference on e-Science and Grid Computing*, pages 177–184.

[Ferreira da Silva et al., 2017] Ferreira da Silva, R., Filgueira, R., Pietri, I., Jiang, M., Sakellariou, R., and Deelman, E. (2017). A characterization of workflow management systems for extreme-scale applications. *Future Generation Computing Systems*, 75:228–238.

[Ferreira da Silva et al., 2015] Ferreira da Silva, R., Juve, G., Rynge, M., Deelman, E., and Livny, M. (2015). Online Task Resource Consumption Prediction for Scientific Workflows. *Parallel Processing Letters*, 25(3).

[Foster, 2006] Foster, I. T. (2006). Globus Toolkit Version 4 - Software for Service-Oriented Systems. *J. Comput. Sci. Technol.*, 21(4):513–520.

[Friedman, 1991] Friedman, J. H. (1991). Multivariate Adaptive Regression Splines. *The Annals of Statistics*, 19(1):1–67.

[Gaussier et al., 2015] Gaussier, É., Glesser, D., Reis, V., and Trystram, D. (2015). Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. ACM Press.

[Gibbons, 1997] Gibbons, R. (1997). A Historical Application Profiler for Use by Parallel Schedulers. *Job Scheduling Strategies for Parallel Processing*, 1291:58–77.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Nets. In *Proceedings of the International Conference on Autonomic Computing*, pages 2672–2680.

[Govindan et al., 2011] Govindan, S., Liu, J., Kansal, A., and Sivasubramaniam, A. (2011). Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, pages 1–14. ACM Press.

[Grandl et al., 2016] Grandl, R., Kandula, S., Rao, S., Akella, A., and Kulkarni, J. (2016). GRAPHENE: Packing and Dependency-aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[Gropp et al., 1999] Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*. MIT Press.

[Gupta et al., 2008] Gupta, C., Mehta, A., and Dayal, U. (2008). PQR: Predicting query execution times for autonomous workload management. In *Proceedings of the International Conference on Autonomic Computing*, pages 13–22. IEEE.

[Gupta et al., 2017] Gupta, I., Choudhary, A., and Jana, P. K. (2017). Generation and Proliferation of Random Directed Acyclic Graphs for Workflow Scheduling Problem. In *ACM International Conference Proceeding Series*, pages 123–127. ACM Press.

[Halzen, 2005] Halzen, F. (2005). IceCube: A Kilometer-Scale Neutrino Observatory at the South Pole. *Highlights of Astronomy*, 13:949.

[Hamilton et al., 2017] Hamilton, W. L., Ying, R., and Leskovec, J. (2017). Representation Learning on Graphs: Methods and Applications. *arXiv:1709.05584*.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*. Data Mining, Inference, and Prediction, Second Edition. Springer Science & Business Media.

[Herman et al., 2000] Herman, I., Melancon, G., and Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43.

[Hindman et al., 2011] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*.

[Hsu et al., 2011] Hsu, C.-C., Huang, K.-C., and Wang, F.-J. (2011). Online scheduling of workflow applications in grid environments. *Future Generation Computing Systems*, 27(6):860–870.

[Huang et al., 2014] Huang, K.-C., Tsai, Y. L., and Liu, H. C. (2014). Task ranking and allocation in list-based workflow scheduling on parallel computing platform. *The Journal of Supercomputing*, 71(1):217–240.

[IBM, 2019] IBM (2019). IBM Spectrum LSF Suites `https://www.ibm.com/us-en/marketplace/hpc-workload-management`.

[Ilavarasan et al., 2005] Ilavarasan, E., Thambidurai, P., and Mahilmannan, R. (2005). Performance Effective Task Scheduling Algorithm for Heterogeneous Computing System. In *The 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)*, pages 28–38. IEEE.

[Ilyushkin and Epema, 2018] Ilyushkin, A. and Epema, D. (2018). The impact of task runtime estimate accuracy on scheduling workloads of workflows. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 331–341. IEEE.

[Im et al., 2015] Im, S., Kell, N., Kulkarni, J., and Panigrahi, D. (2015). Tight Bounds for Online Vector Scheduling. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 525–544. IEEE.

[Iverson et al., 1999] Iverson, M. A., Özgüner, F., and Potter, L. (1999). Statistical Prediction of Task Execution Times through Analytic Benchmarking for Scheduling in a Heterogeneous Environment. *IEEE Transactions on Computers*, 48(12):1374–1379.

[Jain and Rajaraman, 1994] Jain, K. K. and Rajaraman, V. (1994). Lower and Upper Bounds on Time for Multiprocessor Optimal Schedules. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):879–886.

[Jennings and Stadler, 2014] Jennings, B. and Stadler, R. (2014). Resource Management in Clouds - Survey and Research Challenges. *J. Network Syst. Manage.*, 23(3):567–619.

[Juve, 2012] Juve, G. (2012). *Resource management for scientific workflows*. PhD thesis, University of Southern California, University of Southern California.

[Juve et al., 2013] Juve, G., Chervenak, A. L., Deelman, E., Bharathi, S., Mehta, G., and Vahi, K. (2013). Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692.

[Khan et al., 2017] Khan, S., Shakil, K. A., and Alam, M. (2017). Workflow-Based Big Data Analytics in The Cloud Environment Present Research Status and Future Prospects. *arXiv:1709.05584*.

[Klein and Moeschberger, 1997] Klein, J. P. and Moeschberger, M. L. (1997). *Survival analysis: techniques for censored and truncated data*. Statistics for Biology and Health. Springer, New York.

[Köster, 2014] Köster, J. (2014). *Parallelization, scalability, and reproducibility in next generation sequencing analysis*. PhD thesis, TU Dortmund.

[Kougka et al., 2017] Kougka, G., Gounaris, A., and Leser, U. (2017). Modeling Data Flow Execution in a Parallel Environment. In *Big Data Analytics and Knowledge Discovery*, pages 183–196. Springer, Cham, Cham.

[Kougka et al., 2015] Kougka, G., Gounaris, A., and Tsichlas, K. (2015). Practical algorithms for execution engine selection in data flows. *Future Generation Computing Systems*, 45:133–148.

[Krapivsky and Redner, 2001] Krapivsky, P. L. and Redner, S. (2001). Organization of growing random networks. *Physical Review E*, 63(6):066123.

[Kwok and Ahmad, 1999] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471.

[Leipzig, 2017] Leipzig, J. (2017). A Review of Bioinformatic Pipeline Frameworks. *Briefings in Bioinformatics*, 18(3):530–536.

[Li et al., 2019] Li, X., Qi, N., He, Y., and McMillan, B. (2019). Practical Resource Usage Prediction Method for Large Memory Jobs in HPC Clusters. *Economics of Grids*, 11416 LNCS(Chapter 1):1–18.

[Liew et al., 2017] Liew, C. S., Atkinson, M. P., Galea, M., Ang, T. F., Martin, P., and Hemert, J. I. V. (2017). Scientific Workflows: Moving Across Paradigms. *ACM Computing Surveys (CSUR)*, 49(4):66–39.

[Liu et al., 2015] Liu, J., Pacitti, E., Valduriez, P., and Mattoso, M. (2015). A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing*, 13(4):457–493.

[Malawski et al., 2015] Malawski, M., Juve, G., Deelman, E., and Nabrzyski, J. (2015). Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computing Systems*, 48:1–18.

[Mao et al., 2019] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. (2019). Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 270–288. ACM Press.

[Marazzi and Nocedal, 2002] Marazzi, M. and Nocedal, J. (2002). Wedge trust region methods for derivative free optimization. *Math. Program.*, 91(2):289–305.

[Marin and Mellor-Crummey, 2004] Marin, G. and Mellor-Crummey, J. (2004). Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):2.

[Matsunaga and Fortes, 2010] Matsunaga, A. and Fortes, J. A. B. (2010). On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE.

[Miner et al., 2009] Miner, G., Nisbet, R., and Elder, J. (2009). *Handbook of Statistical Analysis and Data Mining Applications*. Elsevier, University of Virginia, Charlottesville, United States.

[Moghadam and Babamir, 2018] Moghadam, M. H. and Babamir, S. M. (2018). Makespan reduction for dynamic workloads in cluster-based data grids using reinforcement-learning based scheduling. *Journal of Computational Science*, 24:402–412.

[Mork et al., 2015] Mork, R., Martin, P., and Zhao, Z. (2015). Contemporary challenges for data-intensive scientific workflow management systems. In *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 4–11. ACM.

[Mu'alem and Feitelson, 2001] Mu'alem, A. W. and Feitelson, D. G. (2001). Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn - Machine Learning in Python. *Journal of Machine Learning Research (JMLR)*, 12:2825–2830.

[Pietri and Sakellariou, 2019] Pietri, I. and Sakellariou, R. (2019). A Pareto-Based Approach for CPU Provisioning of Scientific Workflows on Clouds. *Future Generation Computing Systems*, 94:479–487.

[Powell, 1994] Powell, M. J. D. (1994). A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation. In *Advances in Optimization and Numerical Analysis*, pages 51–67. Springer, Dordrecht.

[Rasmussen and Williams, 2006] Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press.

[Reiss et al., 2012] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. (2012). Heterogeneity and dynamicity of clouds at scale. In *ACM Symposium on Cloud Computing*, pages 1–13, New York, New York, USA. ACM Press.

[Rheinländer et al., 2016] Rheinländer, A., Lehmann, M., Kunkel, A., Meier, J., and Leser, U. (2016). Potential and pitfalls of domain-specific information extraction at web scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 759–771.

[Rocklin, 2015] Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*.

[Rodrigues et al., 2017] Rodrigues, E. R., Cunha, R. L. F., Netto, M. A. S., and Spriggs, M. (2017). Helping HPC Users Specify Job Memory Requirements via Machine Learning. In *Proceedings of HUST 2016: 3rd International Workshop on HPC User Support Tools - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 6–13. IBM Research, Yorktown Heights, United States, IEEE.

[Rodriguez and Buyya, 2017] Rodriguez, M. A. and Buyya, R. (2017). A Taxonomy and Survey on Scheduling Algorithms for Scientific Workflows in IaaS Cloud Computing Environments. *Concurrency and Computation: Practice and Experience*, 29(8).

[Sakellariou and Zhao, 2004] Sakellariou, R. and Zhao, H. (2004). A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. *IPDPS*, 18:1571–1583.

[Schmidt et al., 2018] Schmidt, F., Niepert, M., and Huici, F. (2018). Representation Learning for Resource Usage Prediction. *arXiv:1709.05584*, cs.DC.

[Schultz, 2015] Schultz, D. (2015). IceProd 2: A Next Generation Data Analysis Framework for the IceCube Neutrino Observatory. *J. Phys. Conf. Ser.*, 664(6):062056.

[Schultz et al., 2017] Schultz, D., Riedel, B., and Merino, G. (2017). Pyglidein – A Simple HTCondor Glidein Service. *Journal of Physics: Conference Series*, 898(9):092018.

[Shetti et al., 2014] Shetti, K. R., Fahmy, S. A., and Bretschneider, T. (2014). Optimization of the HEFT Algorithm for a CPU-GPU Environment. In *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, pages 212–218. Nanyang Technological University, Singapore City, Singapore, IEEE.

[Simakov et al., 2018] Simakov, N. A., Innus, M. D., Jones, M. D., DeLeon, R. L., White, J. P., Gallo, S. M., Patra, A. K., and Furlani, T. R. (2018). A slurm simulator: Implementation and parametric analysis. In *Proceedings of the 8th International Workshop on Performance Modeling, Benchmarking, Simulation (PMBS)*, pages 197–217. Springer International Publishing.

[Singh et al., 2017] Singh, A., Rao, A., Purawat, S., and Altintas, I. (2017). A machine learning approach for modular workflow performance prediction. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 1–11, New York, New York, USA. ACM Press.

[Song et al., 2019] Song, J., Li, Q., and Ma, S. (2019). Toward Bounds on Parallel Execution Times of Task Graphs on Multicores With Memory Constraints. *IEEE Access*, 7:52778–52789.

[Sonmez et al., 2010] Sonmez, O., Yigitbasi, N., Abrishami, S., Iosup, A., and Epema, D. (2010). Performance analysis of dynamic workflow scheduling in multicluster grids. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 49–60, New York, New York, USA. Delft University of Technology, ACM.

[Taghavi et al., 2016] Taghavi, T., Lupetini, M., and Kretchmer, Y. (2016). Compute job memory recommender system using machine learning. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 609–616. ACM Press.

[Thain et al., 2005] Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17:323–356.

[Thamsen et al., 2017] Thamsen, L., Rabier, B., Schmidt, F., Renner, T., and Kao, O. (2017). Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference. *BigData Congress*, pages 145–152.

[Thamsen et al., 2016a] Thamsen, L., Renner, T., and Kao, O. (2016a). Continuously Improving the Resource Utilization of Iterative Parallel Dataflows. *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS)*.

[Thamsen et al., 2016b] Thamsen, L., Verbitskiy, I., Schmidt, F., Renner, T., and Kao, O. (2016b). Selecting resources for distributed dataflow systems according to runtime targets . *IPCCC*.

[Topcuoglu et al., 2002] Topcuoglu, H., Hariri, S., and Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

[Tovar et al., 2018] Tovar, B., Ferreira da Silva, R., Juve, G., Deelman, E., Allcock, W., Thain, D., and Livny, M. (2018). A Job Sizing Strategy for High-Throughput Scientific Workflows. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):240–253.

[Tröger et al., 2016] Tröger, P., Brobst, R., Gruber, D., and Mamonski, M. (2016). Distributed resource management application API Version 2 (DRMAA) `https://www.ogf.org/documents/GFD.230.pdf`.

[Tsafrir et al., 2007] Tsafrir, D., Etsion, Y., and Feitelson, D. G. (2007). Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803.

[Tumanov et al., 2016] Tumanov, A., Zhu, T., Park, J. W., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R. (2016). TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 1–16.

[Tyryshkina et al., 2019] Tyryshkina, A., Coraor, N., Nekrutenko, A., and Wren, J. (2019). Predicting runtimes of bioinformatics tools based on historical data: Five years of Galaxy usage. *Bioinformatics*, 35(18):3453–3460.

[Univa Corporation, 2019] Univa Corporation (2019). Univa Corporation - Product Suite `http://www.univa.com/products/`.

[Vavilapalli et al., 2013] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013). Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, pages 1–16. ACM Press.

[Versluis et al., 2019] Versluis, L., Mathá, R., Talluri, S., arXiv, T. H. a. p., and 2019 (2019). The Workflow Trace Archive: Open-Access Data from Public and Private Computing Infrastructures–Technical Report. *arXiv:1906.07471 [cs.DC]*.

[Wang and Peng, 2019] Wang, G. and Peng, B. (2019). Script of scripts: A pragmatic workflow system for daily computational research. *PLoS computational biology*, 15(2).

[Witt et al., 2019a] Witt, C., Bux, M., Gusew, W., and Leser, U. (2019a). Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. *Information Systems*, 82:33–52.

[Witt et al., 2019b] Witt, C., Van Santen, J., and Leser, U. (2019b). Learning Low-Wastage Memory Allocations for Scientific Workflows at IceCube. In *International Conference on High Performance Computing Simulation*, Dublin.

[Witt et al., 2019c] Witt, C., Wagner, D., and Leser, U. (2019c). Feedback-Based Resource Allocation for Batch Scheduling of Scientific Workflows. In *International Conference on High Performance Computing Simulation*, Dublin.

[Witt et al., 2018] Witt, C., Wheating, S., and Leser, U. (2018). LOS: Level Order Sampling for Task Graph Scheduling on Heterogeneous Resources. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 20–30. IEEE.

[Wolstencroft et al., 2013] Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva de la Hidalga, A., Balcazar Vargas, M. P., Sufi, S., and Goble, C. (2013). The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561.

[Yoo et al., 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). SLURM - Simple Linux Utility for Resource Management. *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 44–60.

[Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets - A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[Zhang et al., 2018] Zhang, Q., Kremer-Herman, N., Tovar, B., and Thain, D. (2018). Reduction of Workflow Resource Consumption Using a Density-based Clustering Model. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 1–9. IEEE.

[Zhao and Sakellariou, 2004] Zhao, H. and Sakellariou, R. (2004). An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In *Proceedings of the 9th International Euro-Par Conference*, pages 189–194. Springer Berlin Heidelberg.

[Zhao et al., 2016] Zhao, J., Cui, H., Xue, J., and Feng, X. (2016). Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1443–1456.

[Zheng and Sakellariou, 2013] Zheng, W. and Sakellariou, R. (2013). Stochastic DAG scheduling using a Monte Carlo approach. *Journal of Parallel and Distributed Computing*, 73(12):1673–1689.

List of Figures

2.1. Number of surveys from Table 2.1 that include a workflow management system for comparison against others. Only systems compared in more than one survey are shown. 14

2.2. Example of the HEFT algorithm on two processors p_1 and p_2. The computation times are annotated in red to the nodes as $w(T_i, p_1); w(T_i, p_2)$. The communication times are annotated in blue to the edges as $c(T_i, T_j)/b(p_1, p_2)$; $c(T_i, T_j)/b(p_2, p_1)$. In step 1, average costs are computed. In step 2, task priorities are determined. Step 3 determines the schedule by assigning each task to the processor that minimizes finish time. 19

3.1. Overview of the LOS method and its interplay between exploitation and exploration. Exploitation refers to generating multiple variations of a reference solution by shuffling individual levels, denoted by $r(\cdot)$. LOS prefers those levels for shuffling that have a high estimated probability of finding a better solution. Exploration refers to picking the best variation as a new reference to exploit. 34

3.2. During an exploit step, random variations of a reference schedule are generated by shuffling random levels of the L-Order that corresponds to the reference schedule. 35

3.3. Example dag, its task levels, and an ordering π. The edges of the dag give $T_1 \prec T_2, T_1 \prec T_4, T_2 \prec T_4, T_3 \prec T_4$. Since neither $T_2 \prec T_3$ nor $T_3 \prec T_2$, both tasks can be executed in parallel. The dag has three topological orders: (T_1, T_2, T_3, T_4), (T_1, T_3, T_2, T_4), (T_3, T_1, T_2, T_4), the first two of which are L-Orders. © 2018 IEEE 36

3.4. The level selection process uses the estimated improvement probabilities (red area) to favor promising levels for sampling new variations. The makespans of the L-Orders sampled so far (black arrows on the x-axis) are used to fit a normal distribution for each level. Level 0 has the highest improvement probability, which is reduced by the new solution (bold arrow, dashed distribution). In the next step, level 1 has the highest estimated improvement probability. Note that standard deviations are based on 95% confidence intervals, such that the fitted distributions are much wider when few solutions are available, e. g., for level 2. © 2018 IEEE 39

3.5. Distribution of makespans relative to HEFT's makespan (lower is better) for different numbers of processors and workflow sizes. The baseline method (red) is consistently outperformed by LOS on three processors. The violin plots summarize both the distribution of the relative makespans and their quartiles, as indicated by the three horizontal lines within each colored area. The red reference line indicates 95% of HEFT's makespan. 44
3.6. The average improvement ratio between exploitation phases. Small ratios indicate that solutions are improved using small improvements. 46
3.7. The number of improvements refers to the number of times LOS was able to improve upon its current best solution. The *better* variable refers to whether LOS was able in a given experiment to find a schedule that outperforms HEFT. 46
3.8. Cumulative distribution functions of the latest improvement time, i. e., the elapsed wall clock time when LOS was last able to improve upon its previous solution. Only experiments for which at least one improvement was found are included. 47

4.1. The prediction-scheduling-execution feedback loop. Annotations indicate parameters for the components, such as different task prioritization methods. Dashed lines indicate non-tunable components, compared to the components of the workflow management system for which the parameters can be chosen. 54
4.2. Example data generated using the linear random memory model from Section 4.2.1 with parameters $\mu_y = 10, \sigma_y = 3, R^2 = 0.8, \theta_1 = 0.5, \theta_0 = 1$. The colored lines show predicted peak memory usage as a function of input size for the percentile prediction model (left) and the conservative linear regression model (right). 61
4.3. Mean makespan ratio and memory allocation quality achieved by the 1134 configurations. The right panel is a close up of the area highlighted in the left panel, containing the best configurations. The configurations marked with triangles are Pareto optimal. 68
4.4. Parameter impact on makespan ratio, as measured by averaging the makespan ratios of all simulations using a specific value for a specific parameter. The best and worst values for each parameter are labelled. 69
4.5. Imbalance in value frequency for different parameters among the best 1%, second-best 1%, etc. configurations per workflow. Imbalance is measured as normalized negentropy. For a value of -1, every value is equally often present. For a value of 0, only one value is present. 70
4.6. The frequency of prediction models among the top 5% configurations with respect to makespan ratio and memory allocation quality. Colors indicate the fraction of configurations that use a specific failure handling strategy. 72
4.7. The frequency of basic task priorities among the top 5% configurations with respect to makespan ratio and memory allocation quality. Colors indicate the fraction of configurations that use a specific backfilling parameter k. 73

4.8. Maximum makespan ratio gains from the dynamic configuration scenario. The resulting change in memory allocation quality (percentage points) is displayed on the vertical axis. The red lines mark the average gains. 75

5.1. Memory allocation quality (MAQ) as a function of the first allocation a_{i1} when using exponential re-allocation with different bases. The underlying resource usage measurement set contains a single task $D = (\tau_1 = 1, \tau_1^* = 1, r_1 = 5, x_1 = 0)$. 82
5.2. Cross-sections of the objective function $W(D, \theta, b)$ for three values of b. It shows the memory allocation quality achieved by a rectified linear allocation model as a function of its parameters (slope θ_1 and intercept θ_0). 82
5.3. Schematic of the IceCube workflow. 86
5.4. Total resource allocation per task name. User estimates are rather conservative, strongly preferring oversizing compared to undersizing. 86
5.5. Memory usage as estimated by IceProd users compared to actual peak, median, and interdecile memory usage of the 25 abstract tasks with the highest accumulated run time. Each line shows the four metrics for one abstract task. 87
5.6. Variability of input size and memory usage per abstract task. Regression-based memory allocation has the highest potential where input sizes and memory usage vary strongly (top-right corner) and are highly correlated (red). 88
5.7. Cumulative distributions of memory allocation quality (higher is better) for different prediction models. Memory allocation quality is computed per abstract task, where the first k% (with respect to a job's finish time in the logs) of the data are used for training and the rest for evaluation. The blue line shows the MAQs achieved when applying the peak memory usage estimates provided by the IceProd users. 91
5.8. Cumulative distribution of the differences in memory allocation quality (percentage points) when using LWR instead of the baseline per abstract task. For the vast majority of abstract tasks, LWR performs at least as good as the baseline. Subpar performance occurs on a few abstract tasks of type generate and hits. 91
5.9. Cumulative distribution of oversizing and undersizing wastage (lower is better), relative to the amount of used resources. This shows that the improvements of LWR stem mainly from reducing oversizing wastage. 92
5.10. Effective MAQ when using coarse grained user estimates during training and trained models afterwards. For reference, LWR with a fixed base of 2 is shown, which corresponds to the scenario where the re-allocation strategy is fixed. . 93
5.11. Solutions evaluated by Cobyla, for different initial solutions. The quality of the final solution strongly depends on the starting point of the search. 94
5.12. Memory allocation quality for the solutions evaluated by Cobyla, as shown in Figure 5.11. The initial solution affects both the final solution quality as well as the number of iterations until convergence. 94
5.13. Generating a range of initial solution using the `quantile10` heuristic outperforms heuristics that generate single initial solutions by a large margin. . . . 95

5.14. Chosen slopes and intercepts for the abstract tasks with the most frequent task names. Values are mostly positive and moderate in magnitude. For better display of the outliers, the values have been log-transformed after taking their absolute value and adding one. 96

B.1. Distributions of makespans of L-Orders sampled from single levels of a reference L-Order. Completely random topological orders have a lower probability of outperforming HEFT, which demonstrates the benefit of focusing variations on levels of a dag. The underlying dag (n=50, p=3) was generated with the method described in Section 3.2.1. 115
B.2. Exemplary evolution of level selection probabilities during a single exploitation phase. Note that although level 6 has the highest selection probability most of the time, the randomized level selection scheme makes sure that other levels are also sampled. 115
B.3. Parameter impact on memory allocation quality, as measured by averaging the memory allocation quality of all simulations using a specific value for a specific parameter. The best and worst values for each parameter are labelled. 116
B.4. Backfilling parameter dominance as a function of performance. The negative entropy measures the degree to which a single parameter prevails in a set of results. The figure shows the results partitioned into the best 1%, second-best 1%, etc. The Random heuristic favors certain choices of the parameter, as indicated by high negentropy values. In contrast, LFF is mostly insensitive to backfilling; especially in the mediocre and poor performing simulations, all parameter choices are equally frequent (negentropy close to zero). Note how this also depends on workflow topology; Sipht often deviates from the other topologies. 117
B.5. Optimal model parameters depend on the run times of the tasks as well as the time to failure parameter. On the left side, all tasks are assumed to have the same run time. On the right side, model parameters are optimized for task run times as recorded in the IceCube data. 118

D.1. Example partitioning of a workflow with 9 tasks and specified run times τ. Edges are directed from left to right. 121

List of Tables

1.1. Comparison of the assumptions made in the core contribution chapters. . . . 4

2.1. Surveys comparing scientific workflow management systems and the systems they compare. N denotes the number of compared systems in a survey. . . . 15
2.2. Distributed resource managers supported by Pegasus, Nextflow, and Galaxy. . 22
2.3. Evaluation metrics. f_i denotes the predicted peak memory usage and y_i denotes the actual peak memory usage of the i-th of n tasks in the evaluation data set. $\bar{y}$ denotes the average memory usage, $\hat{L}$ denotes the maximum likelihood of a model, and k denotes the number of model parameters. . . . 30
2.4. Overview of state-of-the-art approaches for predicting memory usage. 31
2.5. Overview of the data used in the memory prediction papers. "Jobs" refers to the approximate number of instances in the training data. "Period" refers to the time period during which the data was collected. "System" refers to the batch scheduler used to run jobs. "Open" refers to whether the data was published or can be obtained from the authors. 32

3.1. Median values of the mean and standard deviation of LOS relative makespans over three runs on each input instance. © 2018 IEEE 45

4.1. The nine basic task prioritization criteria evaluated in the simulation experiments. Tasks with high priority values are started before tasks with low priority values. 57
4.2. Simulation Parameters 65
4.3. Top 3 configurations with respect to average makespan ratio. 99% MSR denotes the 99th percentile of makespan ratio across all workflows. 1% MAQ denotes the long tail performance with respect to memory allocation quality. 66
4.4. Top 3 configurations with respect to average memory allocation quality. 99% MSR denotes the 99th percentile of makespan ratio across all workflows. 1% MAQ denotes the long tail performance with respect to memory allocation quality. 67

A.1. The workflow management system comparison criteria used in the comparison studies listed in Table 2.1. 109
A.2. All workflow management systems appearing in one of the scientific workflow management comparison studies listed in Table 2.1. 111

List of Algorithms

1. The exploitation algorithm samples new L-Orders from the most promising levels as long as the expected time to improvement is below a fraction of the remaining time budget. 41
2. The level order sampling algorithm combines exploration and exploitation into a budgeted search for an L-Order that minimizes makespan. The amount of time spent in the exploit subroutine is chosen randomly (see Section 3.1.3). . 42
3. To decide which task to run next, a priority queue of ready tasks is polled. In addition, a backfilling mechanism allows to skip the top tasks in the queue if the current machine cannot accommodate the predicted memory needs. . . . 55
4. The Low-Wastage Regression approach (LWR) applies the Cobyla method to iteratively refine initial solutions generated with quantile regression. 83
5. The partition algorithm returns a set of task sets such that the sum of the lower bounds on execution time on the induced sub-workflows of the task sets is a lower bound on the execution time of the overall workflow. 123

List of Definitions

Chapter 2

2.1. Definition: Workflow . . . 7
2.2. Definition: Task . . . 8
2.3. Definition: Workflow Language . . . 10
2.4. Definition: Workflow Graph . . . 11
2.5. Definition: Abstract Task . . . 11
2.6. Definition: Path . . . 12
2.7. Definition: Entry/Exit Task . . . 12
2.8. Definition: Level . . . 12
2.9. Definition: Task Lifecycle . . . 13
2.10. Definition: Infrastructure . . . 16
2.11. Definition: Communication Dag . . . 16
2.12. Definition: Allocation Function . . . 16
2.13. Definition: Schedule . . . 17
2.14. Definition: Makespan . . . 17
2.15. Definition: Ordering . . . 18
2.16. Definition: Topological Ordering . . . 18
2.17. Definition: Weighting Scheme . . . 18
2.18. Definition: Upward Rank . . . 19
2.19. Definition: Task Attempt . . . 23
2.20. Definition: Time to Failure . . . 23
2.21. Definition: Resource Usage Measurement Set . . . 24
2.22. Definition: Exponential Re-Allocation Function . . . 25
2.23. Definition: Allocation history . . . 25
2.24. Definition: Wasted Memory . . . 26
2.25. Definition: Used Memory . . . 26
2.26. Definition: Memory Allocation Quality . . . 26

Chapter 3

3.1. Definition: L-Order . . . 35
3.2. Definition: Shuffle Operation . . . 35
3.3. Definition: Region . . . 36
3.4. Definition: Downward Rank . . . 43

Selbständigkeitserklärung

Ich erkläre, dass ich die Dissertation selbständig und nur unter Verwendung der von mir gemäß § 7 Abs. 3 der Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42/2018 am 11.07.2018 angegebenen Hilfsmittel angefertigt habe.

Berlin, den 2. Dezember 2019 Carl Philipp Witt